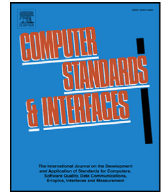
Computer Standards & Interfaces 000 (2018) 1–25

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi



A flexible data acquisition system for storing the interactions on mashup user interfaces

Antonio Jesús Fernández-García[a,*], Luis Iribarne[a], Antonio Corral[a], Javier Criado[a], James Z. Wang[b]

[a] Applied Computing Group, University of Almeria, Spain
[b] Information Sciences and Technology, The Pennsylvania State University, USA

ARTICLE INFO

Keywords:
Mashups
Microservices
Human-computer interaction (HCI)
Data acquisition
Cloud computing
Data management
Data acquisition

ABSTRACT

Nowadays, mashups applications are growing in popularity. They are accessible by cross-device applications, supporting multiple forms of interaction in cloud environments. In general, mashups manage a huge amount of heterogeneous data from different sources and handle different kinds of users. In this respect, mashup User Interfaces are becoming one of the most important pieces in many kinds of current management systems, such as for certain geographic or environmental information systems working on the Internet. In this type of systems, the user interface plays a particular role due to the huge variety of components or apps that the users need to manage at the same time. However, currently, there has been scant attention paid to the management of the user's interaction with mashups interfaces. This goal involves the need of having important, well-constructed tools and methods conducting the data acquisition process for managing properly: (a) the interaction over the mashup user interfaces, at the front-end side; (b) the storage of the interaction in relational databases; and (c) well-supported microservices structures handled in the cloud. The fact of having valuable and flexible data acquisition processes encourages the deployment of others important issues of the interaction management, i.e., data searching, data mining, marketing, security, accessibility, usability or traceability of interaction data, among others. In this article, we present a flexible Data Acquisition System capable of capturing the human-computer interactions performed by users over *mashup* (User) Interfaces with the aim of storing them in a relational database. Firstly, the morphology of traditional *mashup* applications, their specifications and the relevant information that surrounds an interaction have been studied. Thereupon, a data acquisition system that stores user interaction on *mashup* based on such specifications was constructed. To achieve that purpose, an architecture of microservices was also designed in the cloud to detect, acquire, and collect the interactions performed over this kind of interfaces. The whole process is ready for acquiring internal data of the information system as well as context information and location awareness. To validate the data acquisition system, some tests on empirical case studies have been developed. Efficiency and effectiveness have also been determined by evaluating the performance of the acquisition system during different load tests. Finally, in order to ensure the software quality, a continuous integration strategy for software development and an easy management of the code have been used, facilitating the software maintenance alongside the microservice architecture, where functionalities are well encapsulated.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Data is taking an important role in the proper behavior of information systems, since data is becoming a key factor in many projects, offering the opportunity to analyze human behavior in different areas of activity with high reliability by evaluating large-scale data. Specifications of human behavior are not easy to model, not only because of the human nature itself but also because today's information systems can be accessed through heterogeneous devices (e.g., computers, laptops, tablets or smart phones) receiving interaction from different sources such as mouse, keyboard, gestural interfaces or voice recognition (Natural User Interaction, NUI) or emerging technologies, e.g., virtual reality (VR), wearables and Internet of Things (IoT) solutions.

Data sources are widespread in every aspect of our daily life. Click streams from Web visitors, supermarket transactions and operations, readings from all kinds of sensors, video camera footages, GPS trails or

* Corresponding author.
E-mail addresses: ajfernandez@ual.es (A.J. Fernández-García), luis.iribarne@ual.es (L. Iribarne), acorral@ual.es (A. Corral), javi.criado@ual.es (J. Criado), jwang@ist.psu.edu (J.Z. Wang).

https://doi.org/10.1016/j.csi.2018.02.002
Received 24 February 2017; Received in revised form 24 January 2018; Accepted 2 February 2018
Available online xxx
0920-5489/© 2018 Elsevier B.V. All rights reserved.

social media interactions, among others, are examples of the enormous amount of data being generated around the clock. The volume of business data worldwide, across all companies, doubles every 1.2 years [38]. As a consequence, the use of software processes and big data [13] techniques to manage all that data is necessary. In addition, there are certain circumstances around one interaction with the user interface that affect the way humans behave. In particular, we focus our work on the context information. Bearing in mind this context information when designing software to acquire interaction data from the user interface, we can find different kinds of interaction data generated by: (a) the information systems themselves and their users; (b) external third-party components interacting with information systems; or (c) the environment that is acquired by the information systems as context information.

Data management is not always an easy task. Fortunately, today we have tools that make work easier when dealing with data in different stages, from data acquisition to further analysis through data preprocessing. Cloud computing and big data are good examples. Cloud computing focuses on offering us services such as data storage, computing, redundancy, reliability and security for sharing network resources [47]. That makes data available at any time through any Internet-enabled device. Big data is necessary given the need to handle large volumes of data. Generally, such data may become unmanageable, and it may come from different sources and be heterogeneous and complex to process. Furthermore, in many cases, data is generated very quickly, which makes real-time processing difficult.

On the other hand, data acquisition processes [14] are becoming an important task in every information system project, where new standards and methods are emerging to facilitate the information management. The acquisition of data [2,67] using different techniques and tools is common in both social and business environments [39], specially in dynamic context aware application systems. New software architectures or methodologies to model, extract and process data according to some clear specifications have been designed. They include a proper validation of the software developed to that effect, using empirical case studies as close as possible to real cases, to prove the proper performance of the software designed and deployed as well as its quality.

Component-based architecture development is becoming popular and it is being increasingly used in software projects. Although they are appearing new technologies to deal with component-based development such as Polymer, React or Angular, the problem of analyzing the user behavior in these kinds of interfaces is not yet resolved. With the rise of big data and considering that commercial software applications tend to customize user interfaces to improve the users' experience, the topic of extracting the user behavior from these component-based software applications is becoming trendy. For that reason, we think it is important to work on strategies that deal with this issue.

This article focuses on the acquisition of the interaction data of users and mashup interfaces [20,22]. There is an extraordinary potential in analyzing the interaction performed in mashup interfaces. In that sense, the way that the interaction is stored is also important. We pay special attention to graphical user interfaces in this article but mashup application are not only graphical, they are applied in other areas, such as *Internet of Things* (IoT) [30].

As far as we are concerned, we can state that modern mashup software applications lack a proper system to extract, acquire and store the users' interaction with its user interfaces. Our aim is to propose a data acquisition system capable of capturing the user interaction in these mashup interfaces for multiples purposes. Our working hypothesis is that it is feasible to create a data acquisition system capable of storing the users' interaction in a transparent way without compromising their user experience. We can objectively measure this by analyzing the performance of the users' interactions after the deployment of the acquisition system (in seconds) and comparing its performance with others real mashup applications in the market. If the goal of capturing the interaction in a reasonable amount of time without any detriment of the end-user experience is achieved, we can validate the working hypothesis.

In this article, we present a data acquisition system for *mashup* user interfaces in order to extract, receive, manage and pre-process the user's interaction (illustrated in Fig. 1). For that objective, we designed a proper data structure to allocate and store the interaction generated by users in the front-end of the system, i.e., the client *mashup* interface. In this regard, we defined a relational database schema to store the whole interaction data. To validate the data acquisition system, we present an empirical case study ENIA [33], a *mashup* graphical user interface for environmental management of the Andalusia Environmental Information Network [65], a public organization belonging to the Andalusia Regional Government (Spain). Several user profiles and component categories exist in the ENIA interface. Examples of these components include: natural areas, wetlands or biosphere reserves, positioned within maps. ENIA user interfaces are adapted at run-time by applying a model transformation process which is generated dynamically from a set of rules. This process is described in [20] and the goal of the current paper is to improve it from the captured user interaction. Therefore, the purpose of the proposed solution is to provide a flexible data acquisition system which enables a proper mechanism of interaction storage (see Fig. 1).

As far as the authors are aware, there are no similar works in the literature of mashup User-Interfaces that address the problem of capturing the human-computer interactions performed by users over the *mashup* interfaces through a flexible data acquisition system with the aim of storing such interactions in a relational database. In addition, to accomplish this task, an architecture made up of microservices has been designed and deployed in a real cloud environment, making the data acquisition process very flexible.

In summary, the main contributions of this article are the following: (a) Capture human-computer interactions from component-based (mashups) user interfaces distributed in cross-device applications and store them into a relational database; (b) Design and deploy a microservice architecture capable of ensuring the software quality by encapsulating the functionality, and facilitating the software design, implementation and maintenance. We describe a Cloud infrastructure (storage, database, runtime environment, etc.) capable of managing the data acquisition system; (c) Provide a real case study to validate the data acquisition system through deploying the proposed architecture. Some experiments have been designed according to real usage of the applications and load tests have been applied to measure its efficiency and effectiveness.

Once the interaction data is stored in a relational database, the uses of that information are many, for instance:

(a) **Data Searching**. The interaction actions performed by users are always preserved because it represents a way of data persistence. They can be accessed at any time and not likely to be modified.
(b) **Data Mining**. Discovering of patterns in large datasets created from the stored data is highly advisable. Through artificial intelligence, statistics or machine learning algorithms, the user behavior can be analyzed.
(c) **Marketing**. The interaction data can be handled for marketing purposes including promotional activities, analysis, surveys or advertising. Besides, it can also promote new released version of mashup interfaces.
(d) **Security**. The data stored can be secured using Role-Based Access Control as well as provide user identification for every action performed in the information system. Intrusion defense systems (IDS) and intrusion prevention systems (IPS) can be set up based on the storage data.
(e) **Accessibility**. Through data stored, it is possible to study how to develop more accessible mashup designs or to create assistive technology in order to improve the quality of life of certain disabled people having new forms of accessing and interact with *mashup* interfaces.

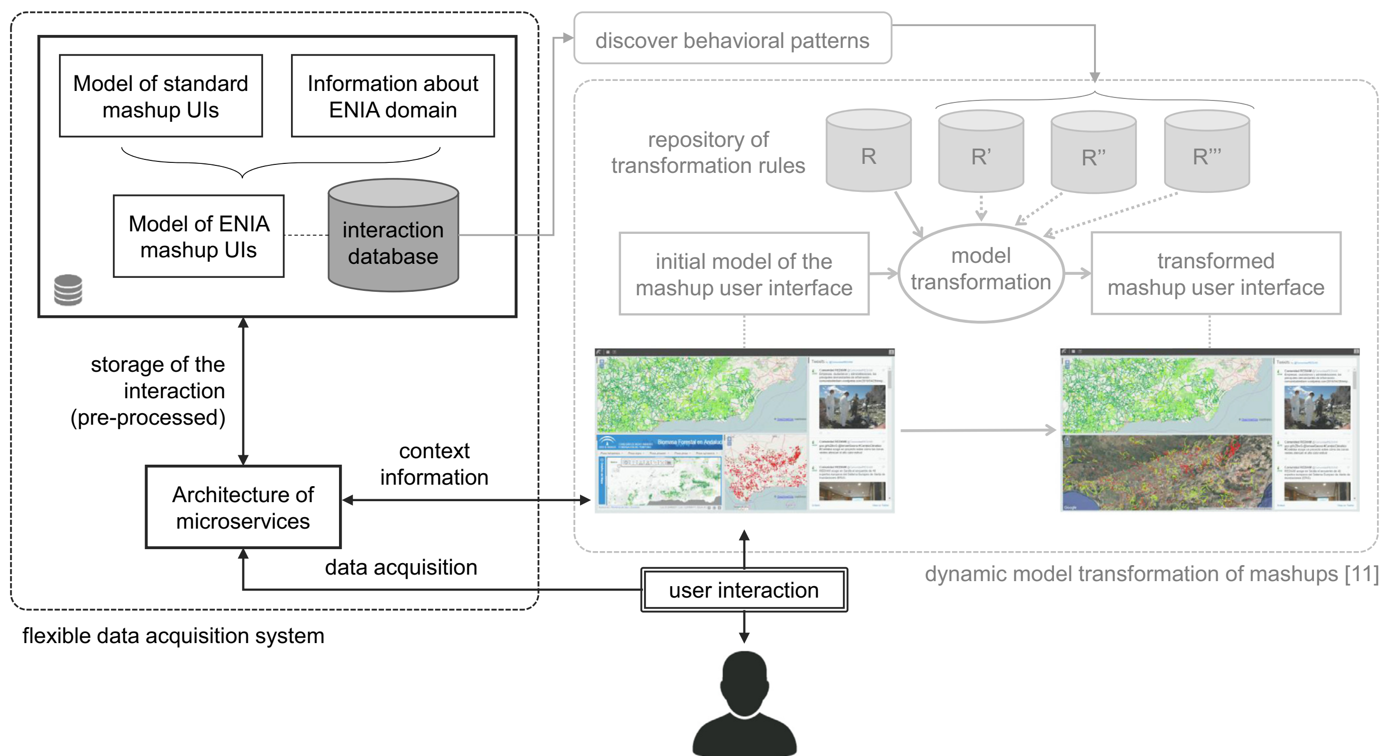


Fig. 1. Proposal of data acquisition to store the interactions on mashup user interfaces.

(f) **Usability**. The stored interaction data can help developers to improve the design and the features of the information systems increasing the effectiveness of the application interacting with users.
(g) **Traceability**. Verify all the steps performed by users when interacting with the information system. It facilitates the ability to chronologically reconstruct the actions performed by users.

The rest of the article is organized as follows. Section 2 reviews some related projects in user interface design and data acquisition. Section 3 describes the morphology of *mashup* user interfaces extended to the ENIA *mashup* case. It also proposes a relational database to store the interaction. Section 4 defines in-depth the proposed data acquisition system and explains the adaptation of this process to the case study. It also discusses a microservice architecture to deploy the data acquisition process. The cloud infrastructure used for the deployment and its scalability rules are described in Section 5. The load tests, conducted to validate the data acquisition process, are fully described and their results analyzed. Finally, some conclusions and further considerations are summarized in Section 6.

2. Related works

The *user interface design* has been explored in the communities of *Human-Computer Interaction* (HCI) and *Software Engineering* (SE). User interface design is a software development task which has been deeply studied from the early years of computer software development [54]. Due to the increasingly complexity applications, the user interface design has become a more difficult task in which many artifacts and software components are required. Many proposals of *software architectures* (e.g., MVC [11], ARCH [8] or PAC Amodeus [19] among others) propose guidelines about how the user interface design, the interaction and the implementation should be carried out. However, the user interface design involves the use of different techniques to model some visual, structural, collaborative, or interaction issues, but also those related to the *acquisition* and *storage* tasks becoming of the user interaction. In addition, user interface design methods involve user participation during the information gathering phases, user interface design and user interface evaluation.

Modeling of user interfaces consists of the specification of the user interfaces through declarative and visual models that describe multiple perspectives and artifacts involved in the development of the user interface (visual, behavior, collaboration, user, interaction, acquisition, storage, among others). There are common kind of models used in most approaches in the literature of user interface interaction development, that are: *task, domain, user, dialogue* and *presentation models*.

Task models specify what tasks the user will carry out on the user interface. Tasks are decomposed in subtasks in order to describe the steps to accomplish a given task. Task models very often include functional requirements and user interface tasks. They also include non-functional requirements like time response. Some of the most well-known approaches for task models are textual methods like GOMS (*Goals, Operators, Methods, Selection rules*) [57] and formal methods like *ConcurTaskTrees* [52].

The *domain model* describes the objects that interact in each task detected in the task model. Although some approaches use the *entity/relationship diagram* [10], most of them use *class diagrams* [58]. Objects can be classified [63] into *interaction objects* (i.e. UI components), and *domain/application objects* (i.e. non-UI objects) representing *databases* or *external devices*.

The *user model* describes user's requirements: preferences, information, devices owned, context, etc. Each user is identified with a *role*. The purpose of the user model is to provide an appropriate user interface for the user requirements. The user model is used to personalize and/or adapt the interface for the user.

The *dialogue model* describes the communication between the user and the user interface. Basically, it describes how the user introduces *input data*, how the user interacts with the user interface, and how the user interface shows *output data*. There are some approaches that use diagrams like *petri nets* [24], *statecharts* [63] or *activity diagrams* [43]. In modern user interfaces like those based on *hypertext* with *dynamic content*, the dialogue model is replaced by the *navigational model* where the user can navigate through different web (*on-the-fly generated*) pages

Table 1
User Interface Development (w.r.t. interaction & acquisition).

Name	Diagrams	Models	Technology
TRIDENT	Entity relationship diagrams and Activity chaining graphs	Task & Presentation Models, User interface Prototypes	WIMP
TERESA	CTT	Task Models, User interface Prototypes	WIMP/Web
WISDOM	Use case diagram, Class diagram, Activity diagrams and CTT	Business, Domain, Dialogue & Presentation Models, User interface Prototypes	WIMP/Web
IDEAS	Sequence diagrams, Class diagram and Statecharts	Task, Presentation, Domain & Dialogue Models, User interface Prototypes	WIMP
ENIA (Our proposal)	BPMN diagrams	Task, Dialogue & Presentation Models, User interface Prototypes	Mashups

following *hyperlinks*. In most cases *statecharts* are used for modeling web navigation [77].

The *presentation model* represents the user interface. It describes the user interface components and its layout. There are some approaches [10,63,76] that offer both an *abstract* model and a *concrete* model. The abstract model describes abstract objects of interaction and the concrete model describes concrete objects for different platforms.

The *Business Process Model and Notation* (BPMN) [72] provides system architects working on analysis and design (A&D) with one well-consistent language for *specifying, visualizing*, and *documenting* the behaviour of software systems, as well as for business modeling. The user interface, as a significant part of most applications, can be modeled using BPMN.

On the other hand, and regarding the level of complexity of the user interface, there are many works in the literature. Most proposal model are WIMP (*Windows, Icons, Mouse, Pointers*) interfaces [3]. However, there are some other proposals [40,77] concerned with the design of modern user interfaces (Post-WIMP interfaces) based on *hypertext* with *dynamic content: client and server side applications*, and *on-the-fly page generation*. There are some works [1,9,12,64,70] in *XML-based representation and manipulation* of *interaction and acquisition data*. Recently, we have the mashup User Interfaces, the focus of the present work. Table 1 summarizes a comparison between these user interface perspectives and our ENIA proposal.

TRIDENT (*Tools foR an Interactive Development ENvironmenT*) [10] is a proposal for an environment of the development of highly interactive applications. User interfaces are generated in an automatic or quasi-automatic way. The task model of TRIDENT uses the *entity/relationship model* together with the so-called *activity chaining graphs*. The presentation model is based on *abstract* and *concrete* interaction objects, presentation units and logical windows.

TERESA (*Transformation Environment for InteRactivE System RepresentAtions*) [9] is a tool designed for the generation of user interfaces for multiple platforms from task models designed with *ConcurTaskTrees*. ConcurTaskTrees (CTT) [52] is a tool based on the formal language LOTOS for analysis and generation of user interfaces from task models. CTT is a formal and visual language for task specification in which *temporal* and *concurrency* relationships can be specified. There are four kinds of tasks in CTT: user tasks, application tasks, interaction tasks and abstract tasks (which are decomposed in concrete tasks). CTT permits the description of complex user interfaces in which multiple users and concurrent tasks can be carried out.

Partially based on UML, WISDOM (*Whitewater Interactive System Development with Object Models*) [58] is a proposal for user interface modeling. WISDOM uses UML but the authors have also adopted the ConcurTaskTrees. The domain model captures the objects and the business model describes each user task and entities involved in such tasks. The business model in WISDOM is represented by the UML use case model and the domain model by a stereotyped class diagram. Use case model is completed by activity diagrams in order to describe the services that the system offers its users. The design model offers a gap between the application level and the user interface. The dialogue model is specified by means of ConcurTaskTrees. In WISDOM there are also interaction and presentation models. The presentation model describes the user interface with stereotyped constructors to describe the structure and navigation through the objects.

Finally, IDEAS (*Interface Development Environment within OASIS*) [51] is also a UML-based technique for the specification of user interfaces in UML and the OASIS (*Open and Active Specification of Information Systems*) specification language. The task model is based on sequence diagrams. The domain model is the class diagram. The dialogue model uses statecharts: transitions can model several windows (dialogue interaction diagrams) and internal transitions in each window. IDEAS can also specify an abstract user interface with components.

Regardless of technology, nowadays, a large number of information systems are interested in acquiring the user interaction through their User Interfaces to learn from them and offer better user's experiences. Most of these information systems require not only data from the inside in order to work properly, but data acquired from the environment analyzed in real (run) time. In most cases, context information helps to understand the human behavior when using a user interface. This context information is collected by multiple sensors or by accessing external web services with related data. The fields in which the data acquisition process to obtains user interaction with human-machine interfaces are vital; we can highlight robotics, medicine, social environments, industrial applications or business, among others. Moreover, their goals are multiple, as can be seen in the related works provided.

In [78] the authors have developed a framework to acquire data from robot skin. As robotics system evolves, they present a standard system where real-time data can be obtained. Its behavior is analyzed, not only for the proper functioning of the robot but with the aim of examining the consequences of its acts. In [42] a scalable data acquisition architecture in Web-Based IoT (Internet of Things) interfaces has been developed. It acquires the data of IoT sensors and also presents a scalable distributed architecture, capable of managing emergencies, which presents a dilemma in system design, in real-time. In [60] the authors conducted a study to identify the important aspects of the user interaction with everyday lighting. This study analyses interaction styles that describe a general way of interacting with a lighting system and present a model that provides the basis for the design of new interfaces.

In other study, an information-rich server logs captured through a web-based platform for participatory transportation planning was used in [71]. The goal was to identify groups of users who use similar patterns and also to evaluate complex software systems beyond the common approach of soliciting user satisfaction. The groups of users were derived through multiple sequence alignment and hierarchical cluster analysis. In works carried out in [23,37,79] the authors studied user behaviors on search engines. In [79] the authors retrieved data from web cache and query logs so that they could optimize the cache data to obtain better results in future searches. In [23] the authors focused on young people because the data acquired by search engines logs show that the results given are significantly less accurate. Finally, in [37] the authors saved data regarding cognitive styles to study how these parameters affect

the results expected by users of search engines with different cognitive styles.

Related to graphical user interfaces (GUI), in [34] the authors used a data acquisition process, adapted in mobile devices, to study the users' behavior in smaller screens; with these data they feed a genetic algorithm that predicts their behavior. In [29] the authors have modeled the behavior of users while using Web Map Tile Services (a tool used to visualize geospatial public data). This type of services consumes a big amount of data when they use geospatial information. The data usage of these services is stored by a data acquisition process for further analysis; thus, they can predict when users are going to require heavy services and anticipate fetching data for those purposes. Due to poor user interface designs or the lack of consistence with the context, in [80] was investigated user interactions over an electronic health records system. Users showed consistent UI navigational patterns, some of which were not anticipated by system designers or the clinic management. Awareness of such unanticipated patterns is used to identify undesirable user behavior as well as create better designs for improving the system's usability.

In some works, such as in [68], the authors presented an approach towards the analysis of users' states in social and human computer interaction based on behavioral cues. The temporal and dynamic integration of their occurrences, and its evolution based on a Markov model sequence, helps to infer the underlying subject state. In other work [31] the author uses some human-computer interaction perspectives to enable user identification by analyzing the user interaction behavior using some indicators such as number of mouse moves, mouse wheel usage and number of keystrokes followed by the number of mouse clicks.

However, as a result of our research there are not in the bibliography any strategy or approach for acquiring the user interaction in user interfaces created using mashup technology as well as we are aware there exists no work that deals with storing that user interaction.

For that reason, we propose a flexible data acquisition system for storing the interaction on mashup user interfaces. As we cannot compare our approach with any existing solution, we validate our proposal by conducting experiments to determine (1) Measuring the data acquisition system performance (including the time of storing the interaction) and (2) Analyzing the mashup application performance compared with other mashup solutions in the market.

According to the mashup user interface literature, there are many examples of commercial component-based interfaces in the industry. Nowadays, mashup interfaces (or graphical component-based interfaces) are widespread in commercial software, particularly in Web applications [32]. Geckoboard [28] is a KPI dashboard surface where users can visualize and work with their most important business data in real-time focusing on sales, marketing or operations among other features. Cyfe [21] allows users to build their own dashboard by adding pieces of information through social media, analytics, sales, finance or project management components, among others. On the other hand, MyYahoo [55] is a customizable page with news, stock quotes, weather, and many other features. Netvibes [56] analyzes and acts on all the data that matters to a brand or business through customized components. Table 2 summarizes a comparative overview of the main features for those mashup applications described above.

There are others examples of mashup user interfaces in the literature; in [45] the authors review some of them, e.g., *iGoogle, Yahoo! Pipes, OpenKapow, Apple Dashboards* or *PopFly*. The authors present a visual composition framework that allow end-users customize applications. In other work, [46], they have studied how end-users with no programming knowledge fail in the customization of their own mashup applications; they studied special factors from industrial applications to incorporate them in mashup applications.

None of the mentioned works propose a strategy for acquiring or storing the user interaction in mashup user interfaces. We suppose that commercial mashup applications may deal with the problem of acquiring and storing the interaction data for their own purposes but if they do, it is not possible to know how they solve the problem because being an internal process, we cannot analyze it.

3. Mashups: User interfaces as-a-service

Mashups applications are a widely accepted strategy for the integration and combination of services. A mashup [32], is a specific type of software that is intended to group services from different sources in the same application. By definition, *Mashup User Interface* [22], is the part of the mashup application that interacts directly with users. As such, it integrates one or more components from one or more sources to create a unique *User Interface* (UI) that combines different components that might (or not) have any relationship among them. There are different integration possibilities, for instance in [44] the authors proposed an architecture for composing enterprise mashup components.

This section explains in detail how mashup user interfaces are composed, with a focus on mashup GUIs (*Graphical User Interfaces*). A standard interface that covers all the common aspects of mashups has been considered. There are many more features available in this kind of interfaces but all of them have some core elements and operations, which have been taken into account in this morphology definition. Next, we will introduce the mashup GUI case study (ENIA), which has many more features that need to be highlighted. Moreover, a morphology definition is also provided for that specific GUI.

3.1. Mashup Morphology: the ENIA mashup case

Fig. 2 conceptually presents a component-based user interface well-known in the User Interface literature as *mashup UI* (to simplify, in some cases is only mentioned as **mashup**). Basically, most of the mashups studied share some core elements [26] as represented in Fig. 2. A mashup contains a set of **services**, which represent the collection of capabilities provided by the application to the users. Generally, services are offered in a **service menu**. Users can navigate through this menu to find the services they may need. Usually, this menu is categorized and grouped by types of services and it has some search tools to locate them directly. A service can be instantiated into a **component** inside a **workspace** in which the users interact with the functionality. Thus, when a user adds a service to the workspace, it is automatically transformed into a component. A component is a service that is being used by a user at a certain moment. When a component is instantiated, a set of attributes (e.g., width, height or position) is assigned to it. Users are allowed to perform some kinds of **operations** over the components, such as resize, move or delete, among others. A more precise study of ordinary mashup morphology is analyzed in [26].

A validation of the data acquisition system is incorporated in this article. For that purpose, the method proposed is currently being applied in a real project called ENIA that will be shown as a case study. ENIA (*ENvironmental Information Agent*) is a mashup component-based graphical user interface for environmental management used by the Andalusian Environmental Information Network [65], a public organization that belongs to the Andalusian Regional Government (Spain). Fig. 3 shows a real screenshot of ENIA *mashup* User Interface[1] that contains four components located in the workspace, namely, two map components, an implementation of a Twitter component, and a weather component. At the left side, the mashup shows a service menu. In the following sentences we will explain the main features of this kind of particular mashup. ENIA possesses all the characteristics of an ordinary mashup user interface [26], i.e., services, components, a workspace, a set of operations and a service menu, and extends its behavior to accommodate other important features that will be explained in the following sentences.

[1] ENIA mashup UI: http://acg.ual.es/projects/enia/ui/.

Table 2
Comparison among mashup applications.

Features	ENIA	Geckoboard	Cyfe	MyYahoo	Netvibes
Free Account			✓	✓	
Payment Account		✓	✓		✓
Add component	✓	✓	✓	✓	✓
Add custom components	✓		✓		
Delete component	✓	✓	✓	✓	✓
Maximize/Minimize components	✓				
Move components	✓	✓	✓	✓	✓
Resize Components	✓	✓	✓	✓	✓
More than 1 dashboard		✓	✓	✓	✓
Share dashboard		✓	✓	✓	✓
Context Information	✓				
Group/Ungroup components	✓	✓	✓		

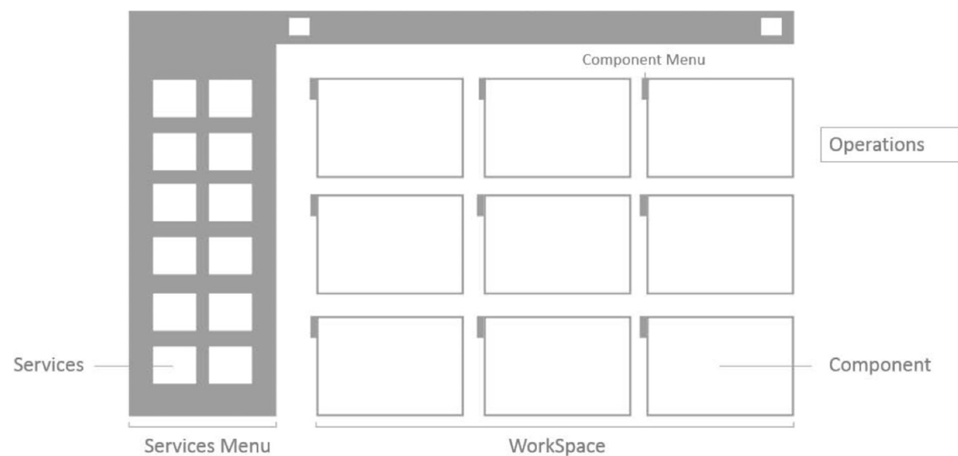


Fig. 2. Conceptual design of a component-based web application.

ENIA is a test scenario with the structure of a common mashup web application but its dashboards and widgets are implemented following specific architectural and component models. These models are described in our previous research works [20,75] as technology-agnostic and platform-independent and they have been validated with W3C widgets in the web domain [75]. Nevertheless, other web technologies such as AngularJS or Polymer can be used to implement these component-based applications if the required constraints and considerations are taken into account (for example, to enable the communication between components through web sockets). As an example of applying the component model approach using Polymer technology can be seen in [5].

COTSget. A notable new feature of ENIA is that two or more components can be grouped together and share their location in the workspace. Thus, it is necessary to create a *container* element that we called as COTSget whose meaning refers to a combination of terms COTS (*Commercial Off-the-Shelf Components*, a way of referring to third-party components) and *gadgets* components (a software artifact that encapsulates the functionality needed to perform a task). This new feature arises from the need to position in the same place data from two different sources, e.g., a map component having real-time information about beach water temperature in Andalusia and another one having information about accessible beaches for people in a wheelchair in Andalusia. It could be useful that in the same map the information of the two components can be combined to facilitate tourists in a wheelchair to visit accessible beaches where the water temperature is nice. COTSgets are graphical component containers. So that, when a component is added into the workspace, the mashup automatically creates a COTSget (container) and the component is placed within it. When a new component is added to the same place as the one occupied by another component, both components will be superposed and will be contained in the same COTSget. One COTSget can contain as many components as necessary provided that the services instantiated have the attribute $groupable = true$. Otherwise, the group will not take place, creating a new COTSget to contain the new component added. Therefore, a COTSget may contain a non groupable component or a collection of groupable components.

Repositories and Origin of the Components. It may seem that `components` are in a single repository but ENIA, like others commercial mashup application agglutinates *components* from different sources. It means that services from the ENIA organization itself and services from others third-party applications can be encapsulated and added to the components repository. Thus, as shown in Fig. 4, the repository does not consist of a particular type of isolated components but a set of encapsulated components from multiple sources. Also, the types of components hosted in the repository are not only web components. The component-based architecture that sustains ENIA, the COScore infrastructure that will be discussed later, support not only web but other types of components that includes, for example, home-automation components.

Group and Ungroup Components. The creation of the COTSget element is due to the need of grouping and ungrouping components. In ENIA, an instantiated service could be groupable or non-groupable. A special type of service in ENIA are OGC (*Open Geospatial Consortium*) [61] services. OGC services have geospatial information that is placed into a map offering the possibility of combining two or more of these services to create elaborate new information. The top panel of Fig. 5 shows a real screenshot of the ENIA mashup UI that contains three components grouped in a single COTSget. The figure shows a base map placed in the south west area of Spain where a component with information

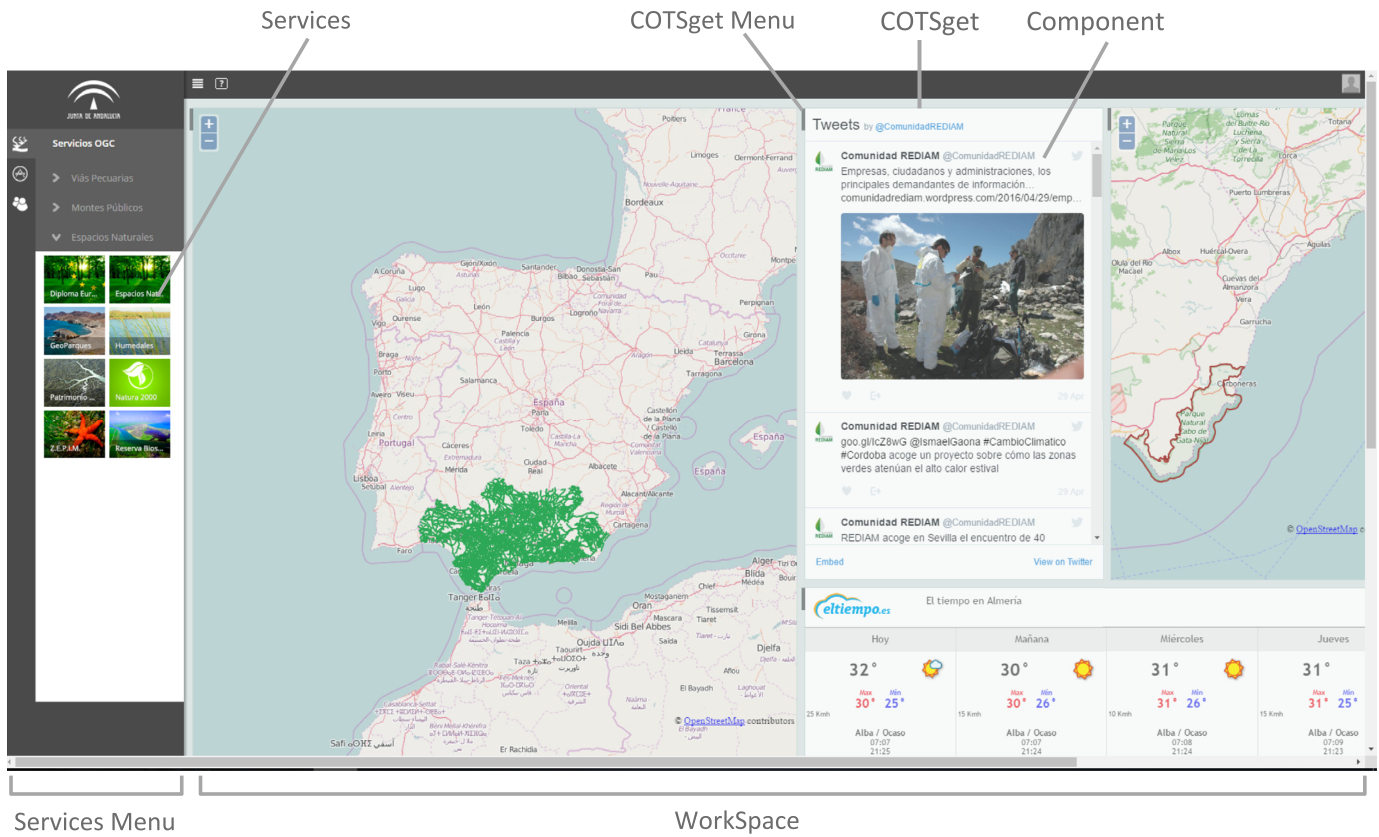


Fig. 3. ENIA mashup UI screenshot.

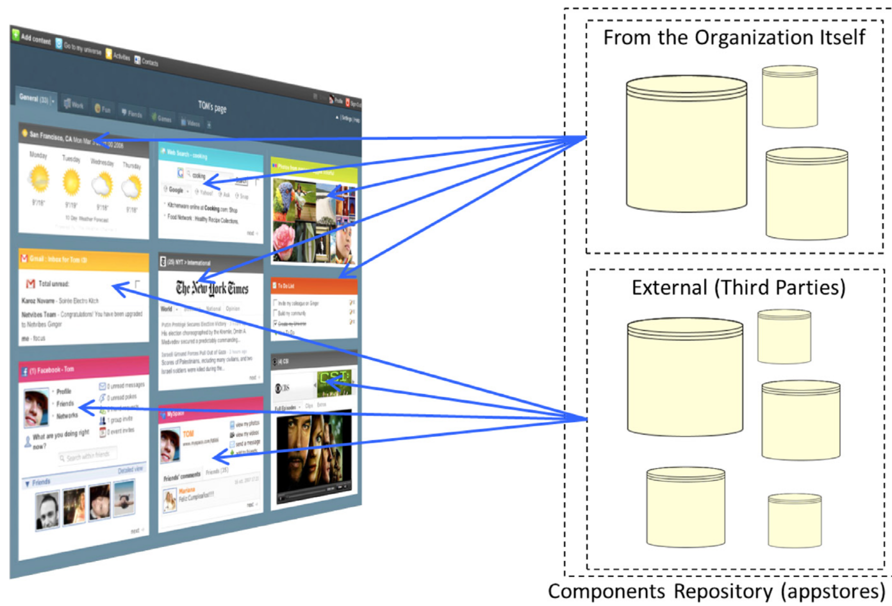


Fig. 4. Repository with `components` from multiple sources.

of geographical parks (red contour) and a component with information of cuttle roads (green lines) are contained in the same COTSget. This kind of overlays can provide useful information. In this example, it is possible to visualize the areas where cuttle roads cross geo parks. The bottom half of Fig. 5 illustrates the same three components instantiated before but, in this case, they are not grouped in the same COTSget. Thus, we can compare the difference between visualizing the same three components apart and grouped in a COTSget as well as we can appreciate the benefits of using COTSgets. The possibility of grouping and ungrouping components leads to the appearance of new operations such as `Group`, `Ungroup`, `AddGroup`, `UngroupGroup` and `UngroupDelete`, which will be further explained in next section.

Multiple instantiation of a service. A special feature of ENIA mashup is that it allows a service to be instantiated several times. This means that more than one component instantiated from the same service can coexist in the workspace. This feature is very useful in some cases.

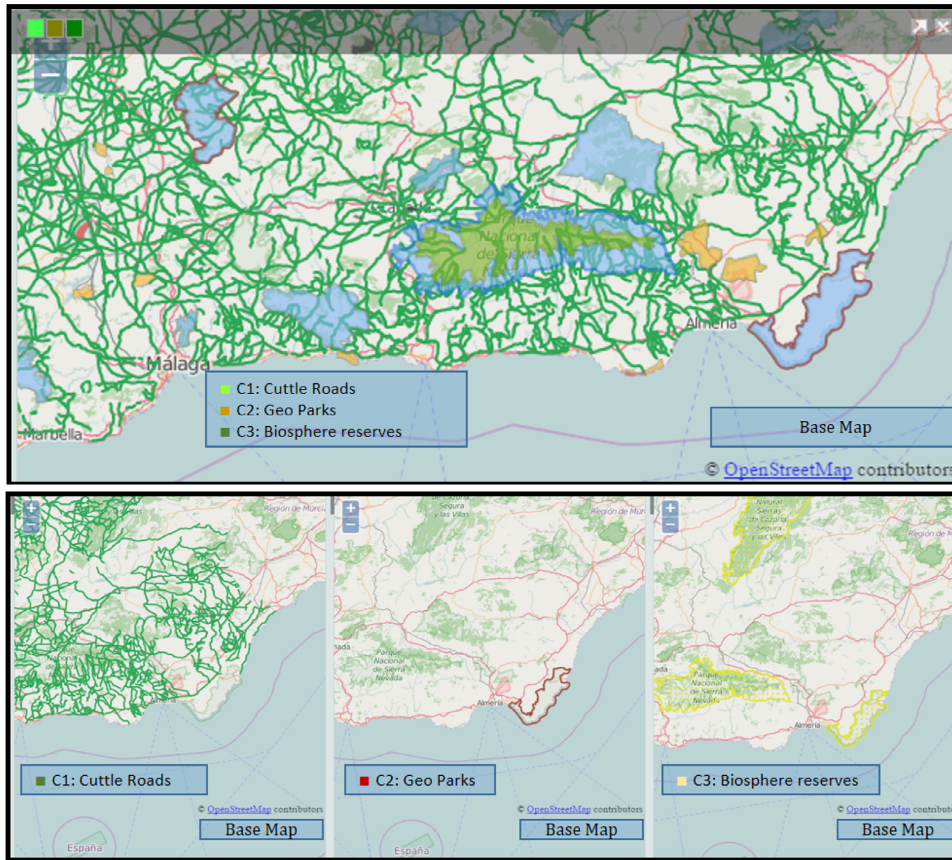


Fig. 5. Workspace with a COTSget that contains three grouped components (top) and the same three components ungrouped (bottom).

Let us suppose that we want to compare the information provided by an OGC service in two different geographic areas. It is easy to instantiate the same service twice in two different COTSgets and place them side by side so that they could be observed at the same time. For example, the "biosphere reserves" service can be instantiated twice, for instance one in the Almeria area (southeast of Spain) and the other one in Seville area (southwest of Spain). As they could be placed together, it would be easy to see their outline and know which one has a bigger biosphere reserve area.

Maximize and Minimize COTSget. The ENIA mashup user interface allow us to maximize and minimize COTSget. Maximize can enlarge a COTSget to be shown full screen. Minimize returns a COTSget that has previously been maximized to full screen to its original size and position. This causes new operations to surface like `Maximize` and `Minimize`, which will be further explained.

Users Information. Registered and guest users can get access to ENIA mashup application. When a user does not identify himself, the system treats him/her as a guest user. In order to be registered, users have to provide some information to the system such as the `name`, `surname`, `birth date`, `address`, `city`, `country` and `email address`. They are also categorized by their user profile in the application. ENIA has two hierarchical levels of profiles: *userType* and *userSubType*. The *userType* profile is the first categorization line and it accepts general kinds of users such as `tourist`, `farmer` or `politician`, all of them being real user types in ENIA. The *userSubType* field goes further in the way of classifying users. It acts as branches of the parent category, *userType*. A user type `tourist` can have user subtypes such as `culture`, `nature`, `health`, `party` or `religion`. A user type `farmer` can have user subtypes such as `irrigation`, `dry` or `greenhouses`. It is possible to dynamically add or delete user types and subtypes profiles.

Services Information. Services are divided according to some properties like `category` and `section`. These properties allow the mashup application to properly categorize the services collection, and help the users to find and use them easily. There are categories such as OGC Services [61], Applications or Social Networks. OGC Services have some sections like Roads or Natural Spaces. The mashup application can dynamically add or delete categories or sections that can be assigned to services. Thus, it is always updated with new services, which are shown to users properly so that they can discover and consume them, emphasizing the most appealing services to users.

Context Information. When an interaction is performed it is linked to some context information. When adapting the data acquisition process to ENIA, it is necessary to collect some context information and save it alongside the interaction data because it can contribute to a better understanding of the interaction produced. Some useful context information considered in the ENIA project has been, for instance, `date`, `time`, `latitude`, `longitude`, `city`, `country`, or some weather description, such as `temperature`, `pressure`, `humidity`, `cloud percentage`, `wind speed` and `wind direction`. This context information can add some value data like the existing temperature in a concrete place-and-time when a user adds a component.

According to these elements above explained, also shown in Table 3, a **mashup** for ENIA user interface ($\mathcal{E}$) is defined as follows:

$$\mathcal{E} = \{\mathcal{S}, \overline{\mathcal{S}}, \overline{\mathcal{C}}, \mathcal{C}, \mathcal{W}, \mathcal{O}\} \quad (1)$$

where,

$\mathcal{S}$ is a set of services $\mathcal{S} = \{S_1, S_2, .., S_n\}$ where n is the number of *services* registered in the information system (catalog of services).
$\overline{\mathcal{S}}$ is a set of menu services $\overline{\mathcal{S}} = \{\overline{S}_1, \overline{S}_2, .., \overline{S}_m\}$ where m is the number of services available in the menu, and $m \leq n$,

Table 3
Elements of ENIA mashup UI.

Element	Description
Services	Capabilities provided by the mashup user interface application. Services are available to users so that they can operate with them. An instance of a *Service* is a *Component*. Generally, services represent external third-party functionalities allocated in a repository in the back-end side (this issue will be explained later in Section 5).
Services menu	In this menu a list of all the *Services* available in the mashup application can be found. Users can navigate through this menu to find the services they may need. In ENIA the *Services* located in that menu are tagged by category, section and subsection.
COTSget	In ENIA, *Components* are not directly located in the Workspace. Instead, they are located in *COTSgets* that can be defined as containers of components. A *COTSget* can contain one or more components with the property $groupable = true$. When a *COTSget* is created a set of attributes like width, height or position are assigned to it.
COTSget menu	*COTSgets* are likely to perform some actions over them. In ENIA, each *COTSget* has a menu placed at the top of it where users can access to a tool bar that displays actions that can be performed over the *COTSget*.
Component	When a user adds *Services* to the *Workspace* they are automatically transformed (instantiated) in *Components* and placed in *COTSgets* container. A *Component* is a *Service* being used by a user at a certain moment.
WorkSpace	The *Workspace* is the work area where users have all the *Components* (*Services* instantiated) they are working with.
Operations	All possible actions that are able to be applied over the `Components` such as add, resize, move or delete, among others.

$\overline{C}$ is a set of COTSgets (containers) defined as $\overline{C} = \{\overline{C}_1, \overline{C}_2, ., \overline{C}_q\}$ where q is the number of COTSget containers created in the workspace $\mathcal{W}$, and $m \leq n \leq q$,

$\mathcal{C}$ is a set of components $\mathcal{C} = \{C_1, C_2, ., C_r\}$ where r is the number of *components* instantiated in the workspace $\mathcal{W}$, $m \leq n \leq r$, and $C_i \sqsubset \overline{C}$,

$\mathcal{W}$ is a workspace where components $\{C_1, C_2, ., C_r\}$ and COTSget containers $\{\overline{C}_1, \overline{C}_2, ., \overline{C}_q\}$ work, and every COTSget $\overline{C}_j$ has some spatial properties s defined as $C_{j,s} = \{x, y, w, w\}$, being (x, y) the position of the component in the workspace, and w and h its width and height.

$\mathcal{O}$ is a set of operations $\mathcal{O}$*={Add, Delete, Move, ResizeBigger, ResizeSmaller, ResizeShape, Group, Ungroup, AddGroup, UngroupDelete, UngroupGroup, Maximize, Minimize}*, which are described as follows:

— *Add operation*. Consists in adding a service to the workspace from the services menu, so it is instantiated into a component, without placing it in an existing COTSget. When instantiating, a new COTSget is created and the new component is placed into it. Some properties such as position in the x-axis, position in the y-axis, width and height are assigned to the COTSget.
— *AddGroup operation*. Add a service to the workspace from the services menu, so it is instantiated into a component, placing it in an existing COTSget. When instantiating the component, it takes the properties previously assigned to the COTSget that contains it.
— *Group operation*. Consists in placing a component that is located alone in a COTSget in other existing COTSget. The original COTSget is deleted because it is empty.
— *Ungroup operation*. Place a component that is located in an existing COTSget with more than one component in a new COTSget that will be created to locate it, hence the component will be the only one that populates the new COTSget.
— *Delete operation*. Consists in removing the last component of a COTSget from the workspace. The COTSget that contains the Components is also removed because it is empty now. More than one component can be deleted at the same time if they are all contained in the same COTSget and the users delete the COTSget itself.
— *UngroupDelete operation*. Delete a component from a COTSget leaving one or more one inside it.
— *UngroupGroup operation*. Consists in placing a component that is located in a COTSget with more than one component into another existing COTSget.
— *Resize operation*. Consists in changing the size assigned to a COTSget and, therefore, all the components contained therein. It modifies the 'width' (w) and 'height' (h) properties. The Resize operation can be decomposed in several operations in the same way as it was previously described for standard mashup user interfaces.
— *Move operation*. Consists in changing the position of a COTSget and all its contained components. It modifies the properties (x, y). When $(x_i \neq x_{i+1}) \wedge (y_i = y_{i+1})$ the component has been displaced horizontally; when $(x_i = x_{i+1}) \wedge (y_i \neq y_{i+1})$ the component has been displaced vertically and finally, when $(x_i \neq x_{i+1}) \wedge (y_i \neq y_{i+1})$ the component has been displaced both horizontally and vertically.
— *Maximize operation*. Consists in increasing the size of a COTSget to show it at full screen. It applies to all the components contained in the COTSget.
— *Minimize operation*. Consists in giving back the spatial properties *width, height*, and position (x, y) to that COTSget that was previously maximized. The COTSget and all the components contained in it go back to their original position and they are no longer shown at full screen.

3.2. Database design for storing interactions in mashups interfaces

Once the morphology of a mashup interface has been presented, let us describe how the interactions are managed. When an interaction occurs in the mashup application, a data acquisition process will get started in order to save all the information regarding the operation by the interaction. Together with the operation, it would be advisable to save the information about the user that generates the interaction as well as the component that is affected and the state that remains in the workspace after the operation. Although it might seem rather costly, storing all the workspace would make it possible to rebuild all the users' behavior step by step throughout the interfaces, in case further analysis, not considered at design time, is required. That is why this option is considered.

In order to store data of the interactions performed by users in the mashup user interface, it is necessary to define a relational database model that is able to store all the relevant information of the interaction. This relational database should be as complete as possible to have a good knowledge of the interaction itself and the circumstances that surround it. Fig. 6 shows a proposed relational database model that stores the interaction performed and its subsequent User Interface representation.

Each row of table `Interactions` represents an interaction taken by the user and the field `operationPerformed` saves the kind of operation performed (add, delete, and so on). The `Interactions` table is related to the `Sessions` table; thus, all operations performed in the same session are grouped. The `Sessions` table has two important fields: `deviceType` and `interactionType`. The first one stores the kind of device used in the session where the operations are performed; it can be a tablet, a laptop, a smartphone, home automation systems or smart watches, among others. The second one (`interactionType`) stores the type of interaction used when performing the operation: mouse, keyboard, gesture or voice, among others. Note that there can be many sessions working currently because

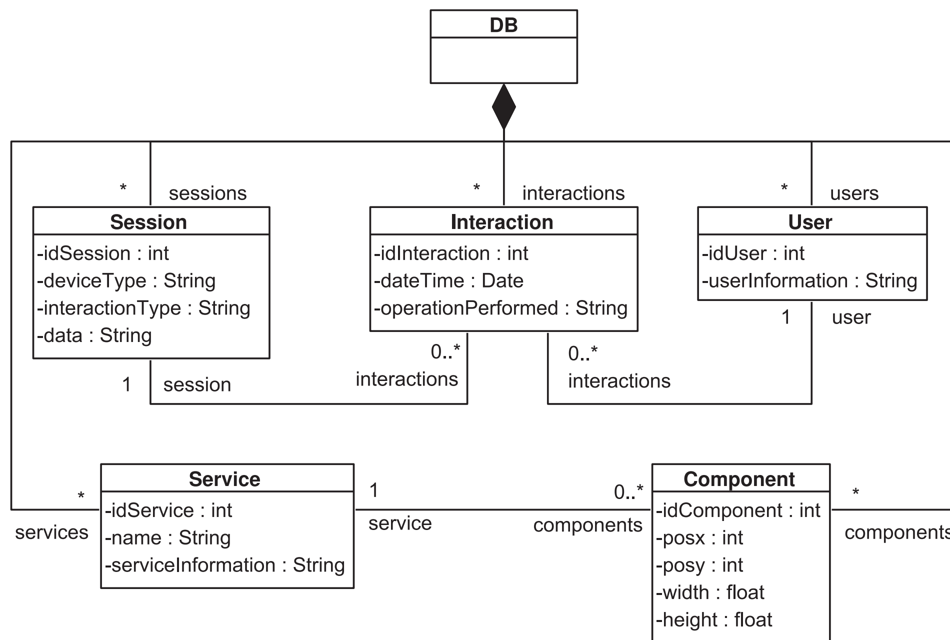


Fig. 6. Database model to store interactions in a standard mashup UI.

a mashup can be used at the same time through more than one user interface from the same or different devices or systems.

Furthermore, the `Interactions` table stores some fields like `temperature` or `humidity` that are stored as *varchar*. This data could be stored in numeric values in order to be more efficient and operate with them. However, considering that these values are obtained from external third-party services, which can be changed unexpectedly, and that the data received can have different formats, we took the decision of storing it in string fields. In the future, this particularity could be addressed when applying feature engineering for data mining purposes, if needed.

The `Users` table, which is related to the `Interactions` table, stores some information of all users registered in the application. Usually, there are lot of users that are registered in most of applications; therefore, it is important to distinguish the operation performed by each one of them. Some information systems allow guest users to access the system and, for that kind of users, there is normally a specific row in the `Users` table. Note that it would be necessary to obtain some extra information about users that does not come with the interaction, such as birth date or user category. Hence, it would be useful to make a request to web services provided by the information system, if any. If the mashup does not handle any users' profile, the `Users` and `Sessions` tables cannot be taken into account.

The `Components` table includes all components that populate the workspace after an interaction has been performed. With the information gathered in this table, it is possible to rebuild the workspace exactly in the same way as it was visually distributed when the interaction was performed. The `posx`, `posy`, `width` and `height` attributes are enough to set each component in the correct position inside the workspace.

Finally, the `Services` table, which is related to `Components`, stores some information about all the services that are registered in the information system. As in the `Users` table, it could be necessary, but not mandatory, to access external web services to obtain more relevant information about services that does not come with the interaction.

This database schema might seem rather costly, since storing a workspace with all its components would involve high levels of consumption of resources. However, in this way, the database is optimized in the sense that it is expressive enough not only to generate datasets with rich data for further analysis (if needed), but to recreate user interaction step by step, in case further information about the interaction could be needed to infer a new knowledge not contemplated at design time.

Fig. 7 shows the design of the database for the ENIA case study (bottom right side of the figure). The diagram also includes the relationship between this model and the ENIA mashup user interface morphology (top right side of Fig. 7). The morphology is represented by using a class diagram in the meta-model level of *Model-Driven Engineering* (MDE) paradigm [6]. Therefore, each operation performed at the instances of this meta-model, i.e., each operation executed in the front-end ENIA interface side, is stored in the aforementioned database model.

Fig. 8 shows the relational database schema that represents an implementation of the model previously described in Fig. 7, in order to store the interaction performed in ENIA as well as the state of the user interface after the interaction. Each row of the `Interactions` table corresponds to an interaction performed by the user. The `operationPerformed` field saves the kind of operation performed and the *dateTime* field saves both date and time when the interaction happened. Moreover, wide range of fields are included to storage the context information. These fields are `latitude`, `longitude`, `city`, `country`, `weatherDescription`, `weatherSubdescription`, `temperature`, `pressure`, `humidity`, `cloudPercentage`, `windSpeed` and `windDirection`.

The `Interactions` table is related to the `Sessions` table, thus, all the operations performed in the same session are grouped. Notice the `deviceType` and the `interactionForm` field in the `Sessions` table. ENIA is prepared to work through different devices with multiple forms of interaction, besides the classical mouse and the keyboard, we can add smartphones, tablets and other devices with gesture or voice form of interaction.

The `Users` table, which is related to the `Interactions` table, stores information of all users registered. In ENIA, guest users are allowed to have access the application. For that kind of users, there is a specific row in the `Users` table that identifies them as guests. ENIA saves extra information about `Users` that should enter in order to be registered in the application. That extra information is: `name`,

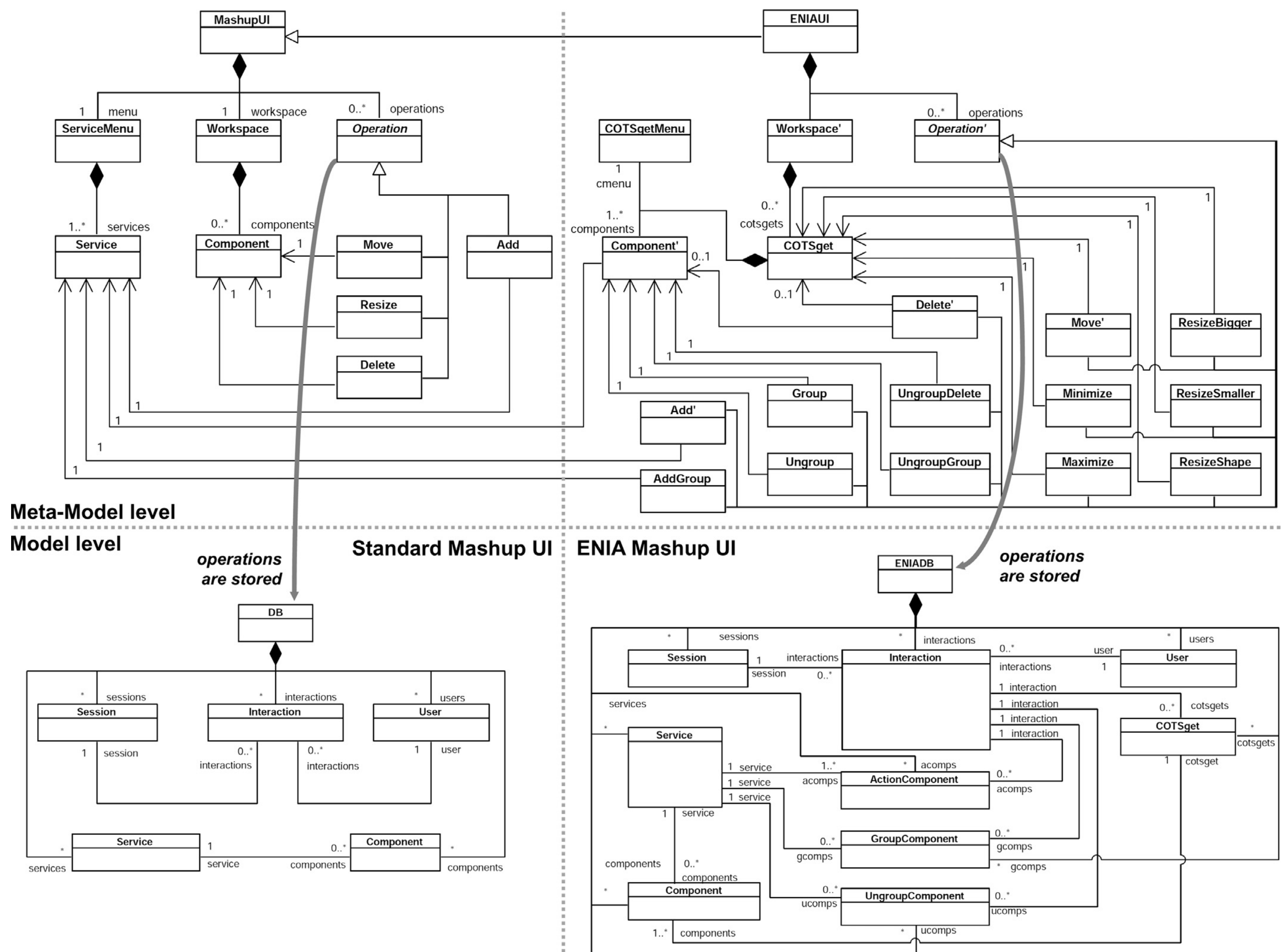


Fig. 7. ENIA database model (bottom right), relation between the model and the ENIA mashup UI morphology (top right), standard mashup UI morphology and its database model (top left and bottom left).

`surname`, `birthDate`, `address`, `city`, `country` and `email`. The `userType` and `userSubType` fields are used to categorize users as it has been previously discussed.

Regarding the protection of users privacy, it should be noted that even when the data is not encrypted the access to it is regulated by an ApiKey as it is described below. Besides, the cloud resources have enabled an IDS (*Intrusive Detection Systems*), Isolated Execution Environments where each individual component (services and databases) run within their own dedicated space and firewall protection that restricts the access to every port from except for those needed and blocks access from not authorized IPs.

The `Cotsgets` table includes all COTSgets that populate the workspace after an interaction has been performed. The `Components` table, which is related to the `Cotsgets` table, stores information about all components in the workspace and indicates in which COTSget they are contained. With the information collected on these tables, it is possible to rebuild the workspace exactly the same as it was before the interaction was performed. The attributes `posx`, `posy`, `width` and `height` are enough to locate each COTSget in the workspace.

The `Services` table, which is related to the `Components` table, stores the information about all services registered in ENIA. The `category`, `section` and `subsection` fields categorize the different kinds of services. That make them easy to find, locate and use in the ENIA UI.

It is important to identify the component on which the operation is performed because this is the component that provokes the interaction. Hosting together more than one component is not an easy task in ENIA due to the use of COTSgets. Sometimes, some operations are not performed on just one component but on a set of components contained in a COTSget. What is more, even when some components are not protagonist in an operation, they can play a second role because they can be directly affected by the operation. For example, that is what happens when a component is grouped in a COTSget with some more components or when a component is ungrouped from a COTSget where other components are left. To store it, we make use of the `ActionComponents`, `GroupedComponents` and `UngroupedComponents` tables, which are all related to the `Interactions` table. The first one gathers the components that provoke the interaction. The second one gathers all components that had previously been in a COTSget before an `Add` or `AddGroup` operation was performed. The last one gathers the components that are left in a COTSget, when an `Ungroup` or `DeleteUngroup` operation has been performed. Table 4 shows the number of components that can be affected in an interaction, depending on the operation performed, thus 0^+ indicates that more than 0 component is affected and 1^+ indicates that more than 1 component is affected.

Table `registeredApiKeys` has no relations with others tables of the database. This table is not necessary to capture the interaction

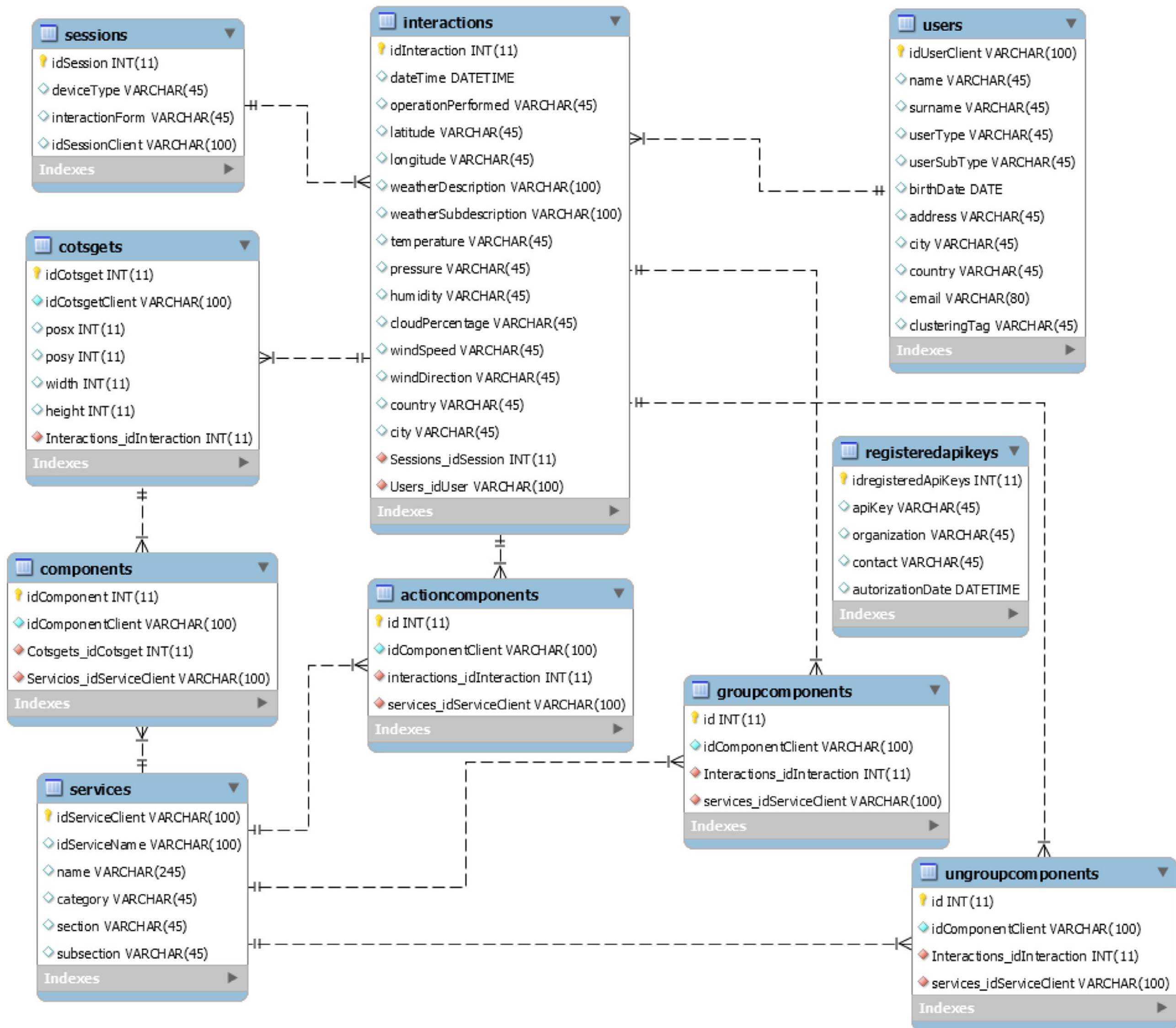


Fig. 8. Database schema to storage interaction in ENIA.

Table 4
Components affected in an interaction vs. the operation (AC: ActionComponents, GC: GroupComponents, UC: UngroupComponents).

Operation	AC	GC	UC
`Add`	1	0	0
`AddGroup`	1	0^+	0
`Group`	1	1^+	0
`Ungroup`	1	0	1^+
`Delete`	0^+	0	0
`UngroupDelete`	1	0	0^+
`UngroupGroup`	1	0^+	1^+
`ResizeBigger`	0^+	0	0
`ResizeSmaller`	0^+	0	0
`ResizeShape`	0^+	0	0
`Move`	0^+	0	0
`Maximize`	0^+	0	0
`Minimize`	0^+	0	0

but it takes a role in the security of the stored data. Fig. 8 (previously shown) represents the real database deployed in ENIA that stores interaction from users. As it is necessary to protect the privacy of users, this table is intended to identify whether the client that sends the interaction information really has permission to do so. Each ENIA client that implements a user interface has assigned an *ApiKey* that identifies it. When such client makes a request to the ENIA data acquisition process, the web service has to submit alongside the ApiKey in order to identify the origin of the request and prevent tampering values.

Note that each table has a numeric auto-increment primary key. In some cases, that numeric primary key could be replaced by a combination of string fields that can provide a primary key. Theoretically, it is better not to have an integer that does not mean anything in relation to the data. But a string primary key with several attributes is harder to compare than a numeric primary key. Besides, primary keys formed from many fields would be substantial to the performance slowdown. Moreover, the size of indexes would be unworkable when the volume of tables increases.

As security measures, we grant access only to authorized users from authorized hosts. When a client tries to establish a connection, we check the username and password and we verify if the connection comes from a host that we allow to connect to the database. If not, the connection is not accepted. Also, each user has different privileges, i.e., users that have only “INSERT” privileges and not “SELECT” privileges, so they cannot even query the database.

Likewise, the data acquisition system can be a point of entry to access private data. For that reason, we have created a web service that only accepts requests from authorized users that can insert interactions by providing parameters through structured data in JSON described in a JSON Schema[2]. The only method implemented in the web service allows inserting interactions in the database following the given structure. There are no methods to delete or to query data from the database. Also, the web service only accepts requests from the host where ENIA is located.

As an additional measure, even to connect to the database through the web service, the clients need a registered ApiKey (stored in the registeredapikeys table), as commented before. The ApiKey is a unique secret token we assign to each client we grant access to the web service. When they make a request, they must submit along with the request the ApiKey so we can verify him and prevent tampering with the values.

The database schema presented here is not optimized for data mining purposes. It is optimized for storing the maximum detail of each interaction performed over the *mashup interface* i.e., to store all the necessary data that allow to rebuild the users' behavior from beginning to end. Our aim is to design a database schema where not only all interactions but also the workspace after each interaction can be stored properly. This makes it possible to recreate the interaction performed by users once again, when necessary, knowing for sure about the UI response and the conditions that surrounded the interaction. In order to apply data mining techniques, it is necessary to previously work with the data stored in this database so that it can be best suited for that purpose.

Thereby, we are not interested in optimizing the database for data mining purposes. We are interested in storing all possible data and in future steps, we will create datasets from this database and other sources that we will use for data mining purposes. These datasets can be optimized through several Feature Engineering techniques such as a) cleaning data to avoid errors and empty values; b) discretizing features; c) creating new features by splitting existing ones; d) creating new features by merging existing ones; e) splitting instances; f) filtering instances; g) adding features from algorithms feedback; h) splitting the dataset. After applying the feature engineering processes enumerated, the datasets will be optimized for data mining purposes.

4. Data acquisition system

Once we have defined the database schema that describes the structure of the database as well as the relational database schema that describes the tables, fields and relationships, this section focuses on the data acquisition system implemented in our approach and a deep description that allow readers easily replicate it.

4.1. Data acquisition methodology

Usually, the data acquisition system has to be integrated into mashup applications through external development teams specialized in data. In the same way, the integration of the data acquisition system is compound of different stages: access to data, extracting data, connect to internal web services, access to external web services, among others. This fact makes that several developers work at the same time in different aspects of the integration. It is for these reasons that we apply agile methodologies for software development facilitating the Continuous Integration (CI) of the data acquisition system in the mashup application. CI practices isolate each process that needs to be implemented providing a secure environment where different developers can work easily. This practice also facilitates feedback between developers (and clients) preventing integration problems.

Through this section we deeply describe the data acquisition system and all the parts that compound it, justifying why agile methodologies are suggested to be followed to implement this approach. The first part will focus on presents a chain of steps with the main parts of the standard data acquisition system and a description of how to deploy it in general scenarios. After it, we deploy the data acquisition system in the ENIA mashup application, providing a case study to illustrate how it can be deployed in real scenarios.

Section 4.2 presents an overview of the data acquisition system and its main parts: provide an interface to acquire the data, gather extra information by querying the client mashup application, access to external context information through third-party services and store all the data gathered in a relational database. Section 4.3 provides guidelines to deploy the data acquisition system in general scenarios by presenting the modules that takes the action and how to deploy them in the own mashup application (modules to detect and extract the interaction) and in the infrastructure deployed to host the data acquisition system (modules for security measures, to validate the interaction, connect to the mashup application web services, connect to third party web services and to communicate with the interaction database). Section 4.4 presents the chain of steps followed to deploy the data acquisition system in the ENIA mashup application case study. A BPMN diagram integrates all the processes implemented and a description of each process detailing how it has been implemented is discussed. Finally Section 4.5 describes the microservice-based architecture that supports the data acquisition system deployed in the ENIA mashup application providing an ultimate vision of the architecture morphology.

4.2. Standard data acquisition process

Fig. 9 shows the steps required to complete a data acquisition process. The first step (#1) is to create a web service to receive any interaction that is produced in the mashup user interface. The mashup user interface client is responsible for calling this web service when an interaction occurs and sends all the basic data to be stored from the interaction. We called this web service `getInteraction`. Usually an XML schema or a JSON schema is provided to the mashup user interface client. This schema describes the necessary data structure to allow the clients to send a document that can be previously validated against the schema to check whether it is 'well formed' and 'valid'.

If we needed any further information about the user, services or any other aspects of the mashup UI, the next step would be to access that data (#2). For that, it is generally necessary to check the mashup application and its information systems to study the web services provided, if any. If so, the request to the web services for the data needed should be implemented in the `getInteraction` web service.

Sometimes there is also some context information that contributes to useful data. When designing the data acquisition process, it is highly advisable to study the environment of the mashup UI and define which data would be worthy to obtain from the context. Usually, access to context information involves connecting with third party web services. As soon as the context data is defined and the services that provide that information are identified, the request for these external web services must be implemented as microservices of the data acquisition web service (#3).

When all these steps are completed and all the required data is gathered, it should be stored in a database (#4). It is required to serialize the database operations in order to avoid integration conflicts when adding relations to the data. Note that relational databases are not mandatory, we should analyze whether or not to use them, and this has to be done for every concrete situation. As previously seen in Section 2, a relational database is used in the methodology proposed in this article due to: (a) Its facility to store structured data; (b) Complex queries can be carried out through the SQL language; (c) Avoid data duplication or inconsistent records; (d) Security control along with access authorization is automatically provided by SQL databases management system.

[2] JSON Schema: http://getinteraction.azurewebsites.net/testbed/jsonschema/index.html.

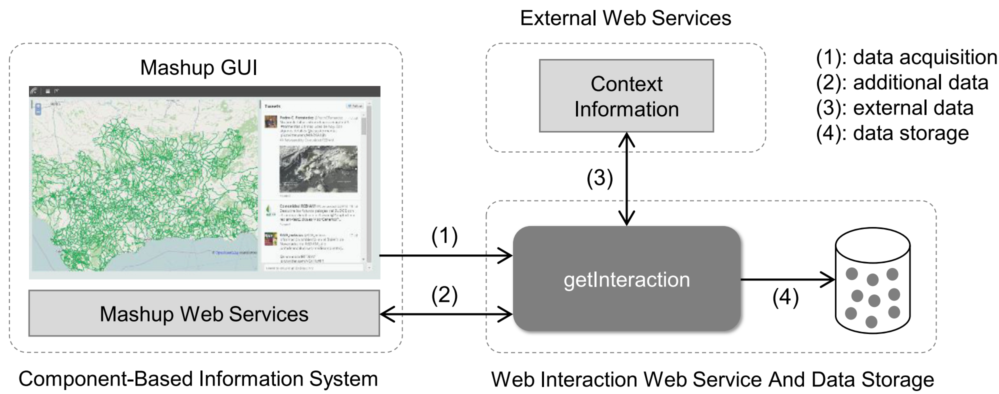


Fig. 9. Steps in the data acquisition process.

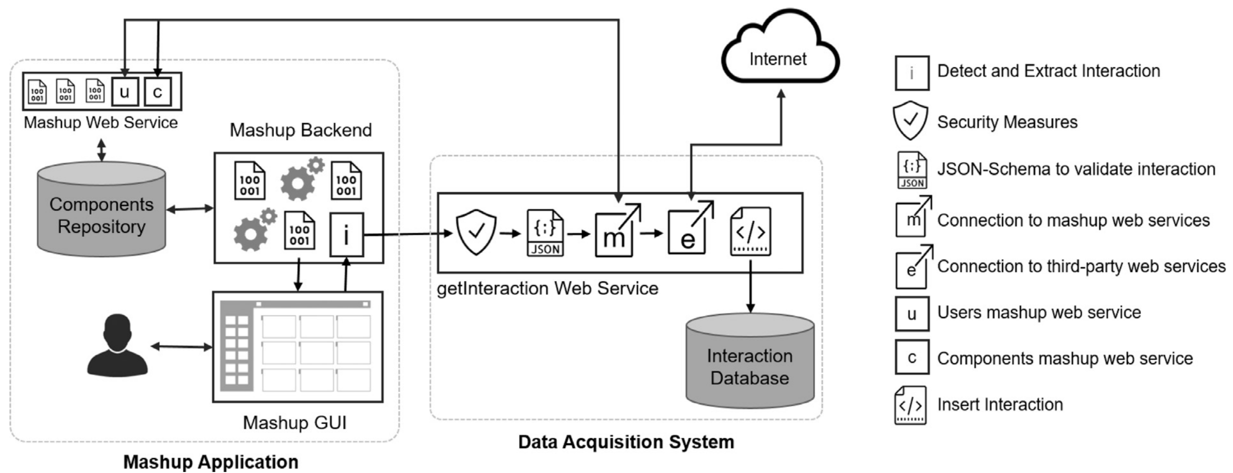


Fig. 10. Data acquisition system and mashup architecture proposed.

4.3. Deploying the data acquisition system in general scenarios

In this section, we provide some guidelines in order to deploy the data acquisition systems in general scenarios. We assume that the mashup application on which we are going to deploy the data acquisition systems has: (a) a component repository; (b) a backend where the logic and data are managed; (c) a GUI where the user interaction takes place; and (d) a web service that allows the mashup application communicating with other services on the Internet. Based on this, we follow a strategy that goes from detecting and extracting the user interaction to storing it in the interaction database. Fig. 10 roughly shows the proposed software architecture and its components at the same time that helps to understand the followed strategy. Two separate parts are visible, the first one is the mashup application and the second one is the needed infrastructure to deploy the data acquisition system. The modifications that have to be made over the mashup applications, as well as how to afford each part of the data acquisition system and its integration with the mashup, are described and enumerated in the subsections presented above. The schematic procedure shown in Fig. 10 graphically illustrates the strategy followed.

4.3.1. Considerations that has to be made on the mashup application

Some considerations has to be made on the mashup application where the data acquisition system is going to be deployed. It is important to know the component-based architecture used to build the mashup application as well as it is necessary to overwrite a method to add a small feature. The considerations are:

(a) Identify the *detect interaction module* (or similar) in the mashup backend and override it. This module, despite being slightly different depending on the component-based architecture and the technology used to build each mashup application, exists in all of them. This is the only part of the mashup application that needs to modify. This module is in charge of receiving a notification when the user interacts with the user interface and adjustments should be made. It has to be modified in order to not just detecting the interaction performed by users in the GUI but extracting and sending it to the *getInteraction* web service.
(b) Mashup web service. It allows communicating with other services on the Internet. Although it is optional, most mashup applications have web services giving more information about their users, components or services. Obviously, no modification should be made on these web services but it is of great interest to know and study them because they can contain useful information that the acquisition system could collect.

4.3.2. Data acquisition system deployment

The data acquisition system, as commented before, consists of a relational database and web service with different modules that process the data received from the mashup user interface after an interaction happen and store it in the database. The modules that made up the data acquisition systems are:

(a) Security measures. Establish the security measures needed to prevent access of unauthorized users from unauthorized hosts and to verify the mashup client authentication to prevent from tampering values.
(b) JSON schema. Check the received data against a JSON schema to verify data is both valid and well-formed.
(c) External sources connection. In many occasions, there can be useful information that does not come with the interaction and it is in-

teresting to collect. We can differentiate two types of data sources information external to the data acquisition system:

— Mashup web services. Connecting to the mashup application web services (if they exist) to obtain internal data that can add more information about users and components is advisable. This data can help to understand better an interaction. If there are no web services available, the data can be exported through any other technique such as CSV, share files, etc.
— Third-Party web services. Connecting to external sources to obtain useful context information (if there are any) that can affect the way users interact with the mashup application is also advisable.

(d) Storing the interaction. Insert the processed and valid interaction alongside other obtained useful information in the interaction database.

In order to easily illustrate the process some graphical representation will be used. For that occasion, BPMN (*Business Process Model and Notation*) diagrams seem appropriate since they are intuitive and easy to understand, yet able to represent complex processes [15]. A comprehensive piece of the BPMN diagram is shown in Fig. 11 and described below.

The acquisition process starts when the mashup extraction interaction module calls the `getInteraction` web service that contains the data acquisition process. When a request reaches the `getInteraction` web service, first of all, the `ApiKey` field is checked. If the `ApiKey` is registered, the client has permission to call the web service so its request is processed. Then, the session is checked. It is useful to group the operations performed by users in sessions, some insights could be obtained by analyzing sessions. Once the session is checked, if it already exists, the database is queried to obtain the `SessionId`; otherwise, a new session is created and inserted in the database.

The next step is to deal with the user processing of the data acquisition process. In this stage the `userID` is checked. If a `userID` exists in the database, that means we already have information from the user; if not, it would be interesting to obtain such information in case the mashups provide it. In the BPMN diagram we suppose the web service exists and it can be seen how the `getInteraction` web service connects to the mashup user web service.

In the diagram we also suppose that it is necessary connects to a third-party web service to obtain external context information that surround the interaction. The figure shows as the `getInteraction` web service connects to a third-party web service and collects the required data. Once the user and context information as well as the interaction information is received, the interaction performed is stored in the database.

After the interaction has been stored, we can distinguish two cases: (1) The mashup application has containers that groups components; or (2) the mashup application does not dispose of such containers. In the BPMN diagram we address the problem where the mashup application dispose of containers (we call them COTSgets), because we want to show the most difficult situation. In case the mashup application does not has COTSget available we can just avoid the next steps and go directly to save the workspace, as we will show later.

Thereupon, if the mashup application dispose of COTSget it is necessary to focus on dealing with the components that are affected by that interaction (`actionComponents`, `groupComponents`, `ungroupComponents`) and how the workspace has been left after the interaction (`COTSget` and `components`).

The `actionComponents` array gathers the components that provoke the interaction and it (the array) needs to be stored in the database. The `serviceID` of each component in `actionCompoments` is checked to find out if the service that instantiate that component has already been registered in the data acquisition process database. If it has been registered, the `serviceID` is taken; otherwise, it is necessary to obtain the information about the service that has been instantiated into the component. If the mashup application has a web service that provides some information about services, it is called.

The same process is repeated for each component in the `groupComponents` array and for each component in the `ungroupComponents` array. The `groupComponents` array gathers components that have been grouped together if an `Add` or `AddGroup` operation has been performed. The `ungroupComponents` array gathers components that have been left together if an `Ungroup` or `DeleteUngroup` operation has been performed.

Finally, in order to save the state of the workspace after the interaction, the list of `COTSgets` (or `Components` if `COTSgets` does not exists) that form the workspace is processed. If `COTSgets` exists, they have one or more components. For each `Component` contained in each `COTSget`. When all the `COTSgets` and `Components` are saved in the database, the process is successfully completed.

In the next subsection, it is shown how this strategy to deploy the data acquisition system is deployed over a real mashup interface: ENIA.

4.4. Data acquisition system applied to ENIA

In this subsection, an implementation of the data acquisition system described in the previous subsection will be applied to our case study ENIA.

4.4.1. Configuration the data acquisition infrastructure

The data acquisition process is hosted in a REST [62] web service that we have called `getInteraction` and has been coded in PHP programming language. We decided to use REST architecture against SOAP [69] protocol, the most standardized, since ENIA is being used in the industry, where REST is more prevalent because of its flexibility and simplicity [18,53]. In general, REST has better performance and scalability, and SOAP requires more bandwidth and resources than REST. Besides, REST is widely used to connect cross-platform applications and even adopted by systems that allows end-user development using them [50].

When the `getInteraction` web service is called, the client sends alongside the request, a JSON with data about the interaction. We use JSON instead of XML due to smaller message size of JSON strings as well as it is can be easily loaded in JavaScript and easier parsed [59]. SOAP only permits XML data-format, and REST permits different data-formats such as plain-text, JSON, XML, HTML among others. A JSON Schema is available to the client. JSON Schema is a powerful tool for validating the structure of JSON data, as an XML Schema is for XML data. It can be used by the client to make automated testing and validating submitted data. The JSON Schema[3] is available in the website that supports the present article.

A BPMN diagram has been built as well for explaining this process. For space reasons and better readability, the whole BPMN diagram is available at the support website[4]. Moreover, the whole source code of the implementation is allocated in Bitbucket[5] available for developers to perform clone or fork actions into theirs projects.

Furthermore, we have created a website that shows how this process works in the real environment[6]. As shown in Fig. 12, this website loads the ENIA mashup UI and overlays a layer on the right side of UI where some information about the user interaction is described in real time. It is important to highlight that the information about the interaction

[3] JSON Schema: http://getinteraction.azurewebsites.net/testbed/jsonschema/index.html.
[4] Data acquisition process BPMN diagram: http://getinteraction.azurewebsites.net/testbed/getInteractionBPMN.svg.
[5] Bitbucket code source implementation: https://bitbucket.org/ajfernandezual/getinteraction/src.
[6] Testbed tool: http://getinteraction.azurewebsites.net/seeInteraction.php.

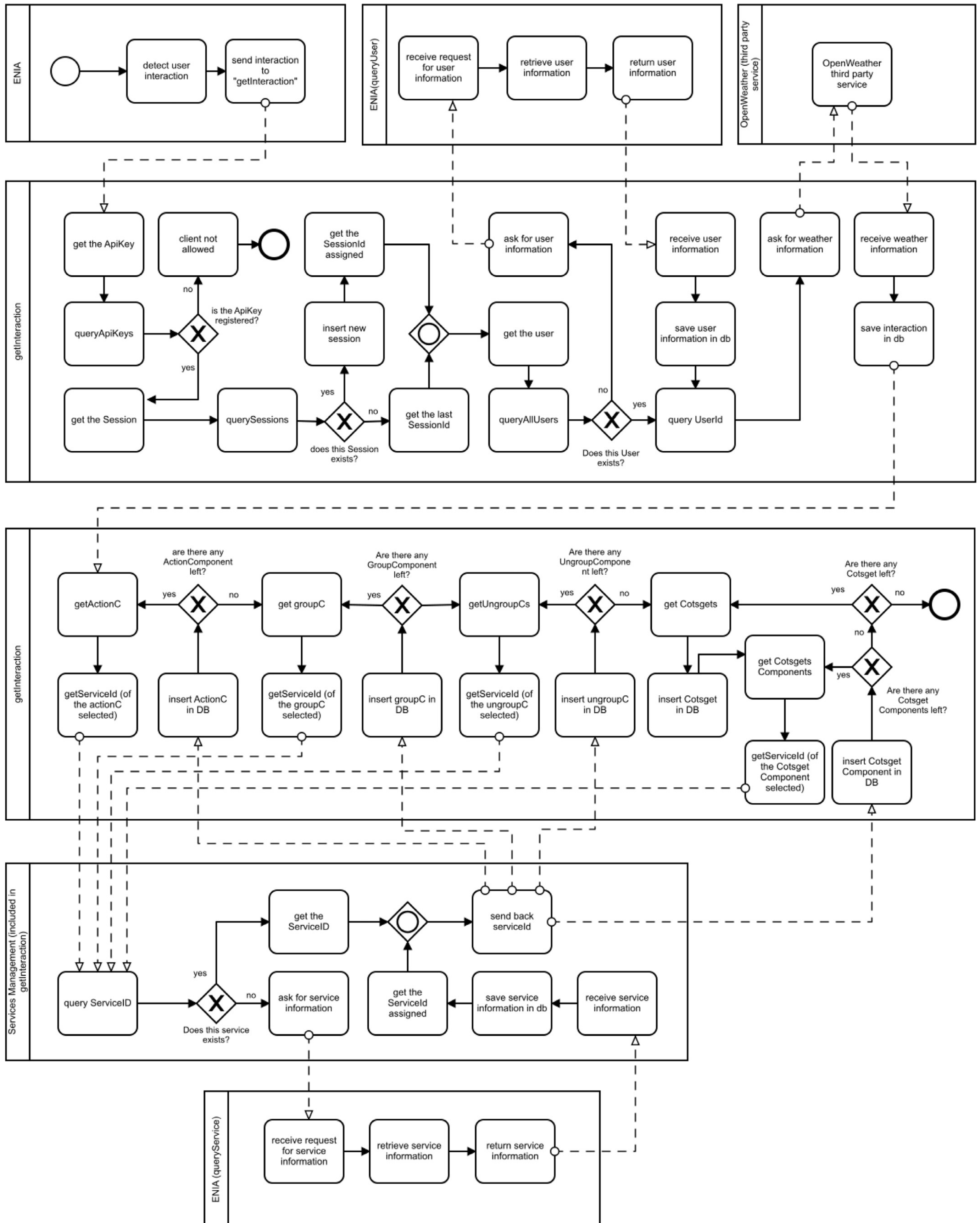


Fig. 11. BPMN diagram of the data acquisition data getInteraction process.

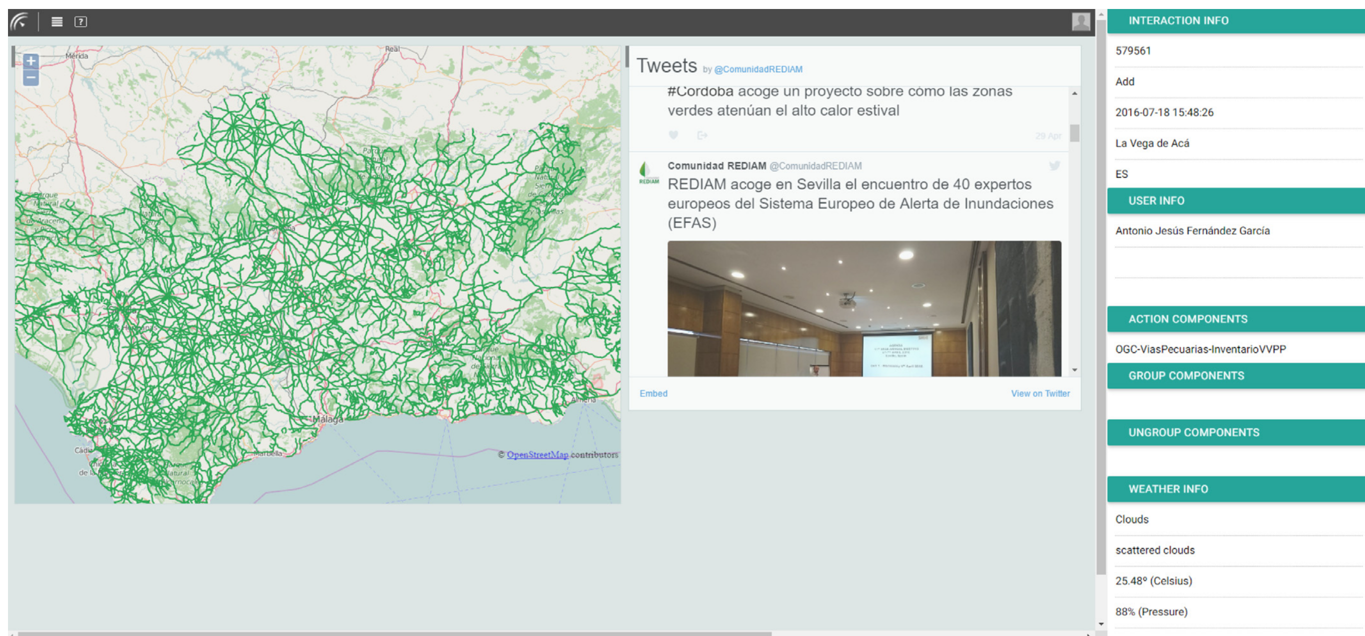


Fig. 12. Website to see the interactions acquired in real time (shown at the right side).

described is not accessed in the ENIA UI but it is accessed through the data acquisition process deployed in the cloud, once the interaction has been processed and stored.

4.4.2. ENIA implementation description

When the `getInteraction` web service receive a request from the ENIA mashup application, first of all, it checks if it comes from a allowed host and the apikey that is sent alongside the request is registered in the database. If the permission is granted, the interaction data is checked in order to know if it is well-formed and valid.

When the interaction data is analyzed, the user responsible for the interaction is analyzed. If there is no information about the user that performs the interaction, the ENIA mashup web services are accessed.

ENIA has a web service that provides information about users: `queryUser`[7]. A request to that web service has been implemented. The parameters of `queryUsers` are the `privateKey` and the `userID`. The return values are `userID`, `name`, `surname`, `email`, `birthdate`, `address`, `city`, `country`, `category`, and `subcategory`. Once this information is gathered, it is stored in the database.

Since the ENIA mashup application works with environmental data, sometimes it is very important for this kind of applications to have information about the weather conditions that surround each interaction performed in the User Interface in certain moment in time. For that, the `ContextInformation` function is called. This function receives `latitud, longitude` as input parameters. Several variables have been regarded as valuable, and the data acquisition process should incorporate them. ENIA does not have such information, so an external third-party API called OpenWeatherMap[8] has been used to obtain the necessary data. Then, a request to OpenWeatherMap has been implemented. The input parameters are: `apikey`, `latitude` and `longitude`. The return values are numerous but we are just interested in: `weatherDescription`, `weatherSubdescription`, `temperature`, `pressure`, `humidity`, `windSpeed`, `cloudPercentaje`, `windDirection`, `city` and `country`. Once this information is gathered, the interaction is stored in the database.

Now, we focus on the components that have been affected by the interaction: (`actionComponents`, `groupComponents`, `ungroupComponents`). In case there is no information about the services that instantiates these components, the `queryService`[9] is called. This web service receives as input parameters an `ApiKey` and a `serviceID`. It returns information about the service to which it belongs. The return values are `serviceID`, `serviceName`, `section`, `category` and `subcategory`.

Finally, we store how the workspace has been left after the interaction (`COTSget` and `components`). For that, a loop runs all `COTSget` that make up the workspace and for each `COTSget` a loop runs all `components` contained and store their size and position.

4.5. ENIA Data acquisition system architecture

The data acquisition system is very flexible, it can be integrated in many ways since it is designed following a microservice-based deployment infrastructure. Each microservice [27,41] is an independent deployable service according with the distributed paradigm where each component in the system is a stand-alone entity that interacts with others across a network with a well-defined interface [35] and it is designed to provide one small well-defined service. The microservices with RESTful web services approach implementation work well and they are being previously evaluated in different works [18].

The data acquisition process has been implemented as a group of microservices due to the following reasons:

(a) Supporting of a wide range of clients such as desktop and mobile browsers, native mobile, tablet applications and even other no graphical interface clients such as gestural, voice and home automation interfaces.
(b) Integration with third-party applications via web services or RPC method for accessing to context information or other kind of information.
(c) Handle HTTP requests and messages, and execute a business logic.

---

[7] ENIA `queryUser` web service: http://acg.ual.es/projects/enia/ui/eniawebservices/.

[8] OpenWeatherMap: http://openweathermap.org/.

[9] ENIA `queryService` web service: http://acg.ual.es/projects/enia/ui/eniawebservices/.

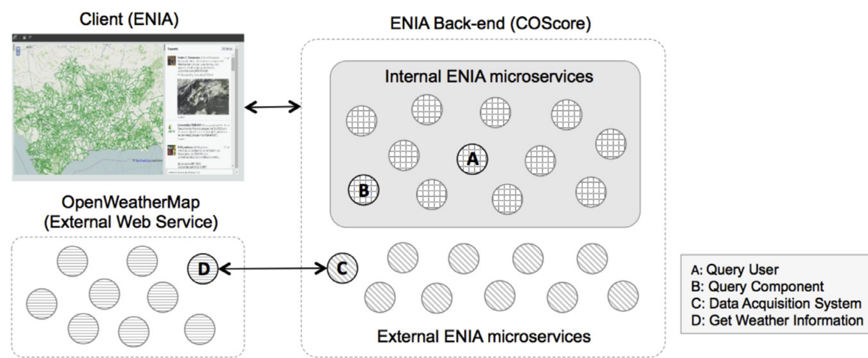


Fig. 13. ENIA `COScore` architecture.

(d) Accessing to the database logic.

The ENIA architecture is shown in Fig. 13. ENIA is powered by `COScore` [74]. The `COScore` is a modular and scalable service infrastructure approach for the management of component-based architectures [20,75]. It provides core services for a mashup UI such as `login`, `logout`, `default init`, `update architecture`, `query user`, `create user`, `delete user`, `update user`, `query profile` among others. The aim of this article is not to describe the `COScore` in detail. Nevertheless, further information about this infrastructure is available in ENIA COScore web services website[10]. This website defines a complete list of internal and external RESTful web services offered by `COScore` as well as the input and output parameter provided with their structure. It also offers developers online service testing functions. In addition, the COSCore Community website[11] provides general information about the `COScore` infrastructure community, and GitHub repository support for open source versions of COScore.

Over the ENIA infrastructure a new external microservice has been added, that is, the data acquisition system described in this article. This microservice consumes some ENIA web services, specifically, the `queryComponent` and `queryUser` services to obtain internal information of ENIA components and users. It is also connected with third-party services, particularly, with the `getWeather` of the OpenWeatherMap Rest API. Finally, this microservice stores the collected data in the database.

5. Infrastructure, technologies, performance and validity threats

This section describes the infrastructure and technologies that have been used to implement the data acquisition system. This is the basis for monitoring workloads and for analyzing the performance of the mashup application in real situations.

5.1. Technologies and infrastructures

The data acquisition system has been coded in PHP 5.4 programming language. No development environment has been used, just Notepad++. For the code management, we have used Git and we have integrated it with Bitbucket as the code repository of the Git version control system. We have used SourceTree Git desktop client to sync the code to the repository. Both Bitbucket and SourceTree belong to Atlassian software enterprise.

The data acquisition system runs in the cloud. The service provider that hosts it is Microsoft Azure® under a platform as a service cloud layer. Azure provides native continuous integration with Bitbucket[12].

[10] ENIA COScore API: http://acg.ual.es/projects/enia/ui/webservices/.
[11] COScore Community: http://acg.ual.es/projects/enia/ui/coscore/.
[12] Git Repository in BitBucket: https://bitbucket.org/ajfernandezual/.

Table 5
Performance metrics that set scalability rules for the cloud resources.

Metric	Threshold	Duration	Action
CPU Percentage	>=80%	30 m	increase 1 instance
CPU Percentage	<=60%	30 m	decrease 1 instance
Memory Percentage	>85%	10 m	increase 1 instance
Memory Percentage	<65%	10 m	decrease 1 instance
HTTP Queue Length	1000 request	5 m	increase 1 instance
HTTP Queue Length	100 request	5 m	decrease 1 instance

The database, also hosted in the Azure cloud, is a relational MySQL® database supported by ClearDB: a reliable, fault tolerant database as a service. MySQL Workbench from Oracle provides data modeling, SQL development and administration tool to manage the MySQL database hosted in ClearDB.

All these tools provide a useful environment to apply a continuous integration strategy for software development and an easy management of the code, data and database.

Both the computer instance hosting the data acquisition system and the database that stores the interaction, respect the SOLID interface segregation principle (ISP), i.e., they are isolated preventing coupling to dependencies. The Azure D-series instances used are based on the 2.4 GHz Intel XeonÂ® E5-2673 v3 (Haswell) processor generation [7]. The ClearDB MySQL database allows up to 15 connections and a size up to 10GB. Its redundant architecture has a 99.99% up-time as well as a geo-distributed topology with no-downtime scaling and automatic failover [16].

To setup the data acquisition, we defined an instance in Azure with 1 Core and 1.75 GB Memory located in West US. This instance is always up and running. At certain times, there might be peak loads that could compromise the handling of HTTP Requests (containing interaction data) that can reach the server. Fortunately, cloud providers allow adjusting the resources at real time, enabling the system to scale up the usage of resources, when necessary; or scale it down, idle resources exist [48].

Based on some performance metrics a set of rules has been defined in order to manage the scale up and scale down of resources. An instance with 1 Core and 1.75 GB Memory is always up and running, and a maximum of three instances with exactly the same characteristics can be used, if necessary. The scale-up or scale-down is performed, given the rules defined in Table 5. CPU Percentage, Memory Percentage and HTTP Queue Length are the metrics chosen to determine if a new instance should be added or removed. CPU and Memory are measured in percentage. When there is more than 1 instance running, the percentage of CPU or Memory used correspond to the average of all the instances running at that time.

We set the scale-up to one more instance when: (a) the CPU percentage is above 80% during more than 30 min. (we avoid to scale-up in a

Table 6
Operations performed for storing 1 interaction.

Part of the Process	Read DB	Write DB	Request
Apikey	1	0	0
Sessions	1	0 or 1	0
Users	1	0 or 1	0 or 1
Interactions + Context	1	0	1
Action Components	0	a	0
Group Components	0	g	0
Ungroup Components	0	u	0
COTSget	0	cg	0
Components	0	c	0
Services	c	0 to c	0 to c
Total (Pessimistic)	4+c	2+a + g + u + cg + 2c	2+c
Total (Optimistic)	4+c	a+g + u + cg + c	1
Total (Average)	4+5.4=9.4	1.4+0.6 + 0.4 + 3.2 + + 2*5.4 + 0.25=17.65	1.2
Performance Test	4+6=10	1+1 + 0 + 3 + 2*6 + 1=18	1

shorter period of time because the high CPU percentage can occur due to internal processes of the operating system no related to the data acquisition system); (b) the memory percentage is above 85% during more than 10 min. (it is an indicator that the memory is busy with the data acquisition system and no to other processes); and (c) the HTTP Queue Length has more than 1000 requests waiting to be attended during more than 5 min. (it indicates that the system cannot process properly the current workload).

We set the scale-down to one less instance when: (a) the CPU percentage is below 60% during more than 30 min. (the CPU is not being compromised anymore); (b) the memory percentage is below 65% for more than 10 min. (memory is not being compromised anymore); and (c) the HTTP Queue Length has less than 100 request waiting to be attended for more than 5 min. (it indicates that the requests are being attended properly).

It is important to highlight that the scale-up is performed just when one of the scaling rules is met and the scale-down is performed when the three rules are met together.

5.2. Performance study of the interaction

In order to evaluate the performance of the data acquisition system let us describe the operations that computationally perform the process. Once an interaction occurs, two major kinds of operations are executed: (1) *read/write* database operations and (2) third-party requests to external web services. Table 6 shows each part of the data acquisition and its implications.

When calculating the total number of *ReadDB, WriteDB* and third-party *Request* operations, an estimation of the average number of these occurrences that happen in the interactions has been made. Based upon previous activity registered in the database by real users through the implemented data acquisition system, we estimated that the average of components that exists in the workspace when an interaction is performed is 5.4, the average of COTSgets is 3.2 and so on. Thus, we established that an average of 9.4 *read* operations, 17.65 *write* operations and 1.2 third party HTTP request are performed for each interaction. This may cause a substantial computational load and resources usage, so the concern about whether or not it is feasible arises. For that reason, some performance tests are conducted to determine if the resources provided to this effect are enough.

It would be really interesting to carry out these experiments in other mashup application to make a comparative. Unfortunately, it is not possible due that we do not even know whether the other mashup applications analyzed in the related works store the user interaction or not. Even assuming they do, it is not possible to obtain data about the process because it is an internal process to which we do not have access. It would also be interesting to compare our data acquisition process for

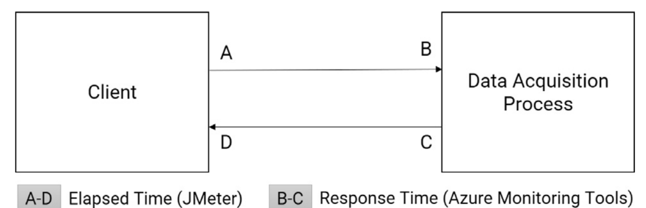


Fig. 14. Elapsed time vs. response time.

mashup applications to others analogous processes previously discussed in the bibliography but, as far as we are aware, it does not exist a similar process for this purpose. This is another reason why we consider relevant this work.

Taking that into consideration, a number of three experiments have been carried out (explained later). In such experiments, according to the usage statistics shown previously, some requests to the data acquisition system were modeled so that they could adapt as well as possible to real situations. The aim of these experiments was to find out if the data acquisition system was ready to bear its workload. Each experiment was carried out with a load test. We measure and analyze the results of the experiments from two perspectives: (a) *Apache JMeter* and (b) *Azure Monitoring Tools* (AMT).

Apache JMeter [36] has been used to analyze and measure the data acquisition system, by making the requests to the system and obtaining data. A request uses 10 *read* operations, 18 *write* operations and 1 HTTP third-party web service request. This is the closest to an average request to the data acquisition process. From JMeter, we obtain external performance measures such as `Average Elapsed Time` (an average time since a request is sent from JMeter and an answer is received indicating that the request is processed), `Median` (separates the higher half of elapsed times from the lower half), percentile `95%Line` (higher elapsed time of the 95% lower elapsed times), `Min` (Minimum elapsed time) or `Max` (Maximum elapsed time).

Azure Monitoring Tools (AMT) has been used to obtain internal performance measures of the cloud infrastructure resources such as `Average Response Time` (average time since a request is received until it is processed), `CPU Time` (percentage of the CPU time available dedicated to process the request) or `Average Memory Working Set` (average amount of memory consumed to process the requests).

From the experiments point of view, it is important to clarify the difference between elapsed time and response time. Fig. 14 helps us to understand that difference. The elapsed time is the spent time when the client side sends a request until it receives an answer from the cloud, that is, the request time from the client to the cloud, the response time of the cloud, and the answer time from the cloud to the client. It is the

Table 7
JMeter experiments results.

Requests	Average	Median	95% Line	Min	Max	Error
30	477 ms	321 ms	1018 ms	266 ms	2818 ms	0.00%
1800	6968 ms	7579 ms	14955 ms	23 ms	23869 ms	0.00%
6000	13743 ms	13580 ms	30317 ms	18 ms	50068 ms	0.00%

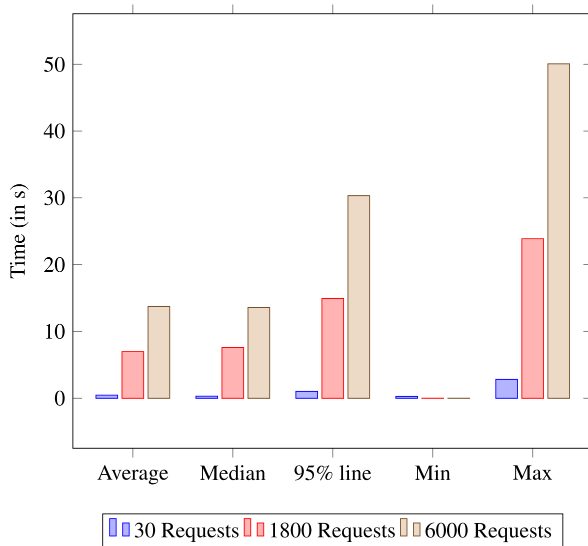


Fig. 15. JMeter experiments results.

time between A and D shown in Fig. 14, and we get that measure from JMeter. The response time is the amount of time spent when the cloud receive a request until the cloud sends an answer. The response time does not include the amount of time since the client sends the request until it arrives to the cloud nor that the cloud sends the answer to the client. It is the time between B and C shown in Fig. 14, and we get that measure from Azure Monitoring Tools.

Regarding the experiments, they were considered according to the number of hypothetical real users interacting with ENIA mashup. In the first experiment, we considered a load test for the REDIAM's staff using ENIA. As we advanced, REDIAM is a governmental institution of the Andalusia region (south of Spain). The aim of REDIAM is the production, exploitation and management of large environmental and geographic information systems. For a second experiment, we defined a load test where we supposed that all the environmental staff in the Andalusia region will access to ENIA. In the third experiment, we modeled a load test in which we supposed that ENIA will be accessible by all the tourists that visit Andalusia every year. Since we had no real data for the second and third case, an estimation of users were made (later justified).

5.2.1. *Experiment #1: REDIAM Staff*

At present, about 10 users in the REDIAM agency are concurrently working with the ENIA mashup web application, with an average session time of 30 min.. Each user approximately performs an average of three operations in one minute. This means that 30 requests will be made per minute or one request every two seconds. Therefore, when setting up the JMeter test a new thread group is created and parameters *number of threads* and *ramp-up period* are set to 30 and 60, respectively. The *number of threads* refers to the total number of requests that will be made, and *ramp-up period* is the time taken by JMeter to start (to create and send) the requests defined.

The results of this experiment can be seen in the first row of Table 7 and graphically illustrated in Fig. 15 marked in blue. Internal data obtained with Azure Monitoring Tools can be seen in the first column of Table 8 and graphically illustrated in Fig. 16.

Considering the outcomes of JMeter, in this experiment we can observe an average elapsed time of 477 ms, which is acceptable. Better yet, the median throws 321 ms. The `95%Line` shows a value of 1,018 ms which means that 95% of the requests are processed in less than approximately 1 s. The rate error is 0%, and therefore, the data acquisition process has processed all the request received correctly. When analyzing data from Azure Monitoring Tools, we can observe an average memory working set of 32.75 MB, which is pretty low, and a CPU Time of 0.92 s, which represents the 1.53% of the time available. The average response time is 413 ms, which is logical since the elapsed time is 477 ms and the remaining 64 ms is used in processing communications.

5.2.2. *Experiment #2: Andalusia environmental staff*

For the second load test, we considered the number of users that would access to the ENIA mashup, whose jobs were related to the Environment topic in the region of Andalusia and employed by the regional government. On the basis of [17] we estimated that ENIA mashup would be used by 600 users and the average session time would be 30 min., where each user approximately performs an average of 3 operations per minute. This means that 1800 requests will be made per minute, i.e., 30 requests per second. When setting up the JMeter test, a new thread group is created and parameters are set to 1800 for *number of threads* and 60 for *ramp-up period*. The results of this experiment can be seen in the second row of Table 7 and graphically illustrated in Fig. 15 (in red). Internal data obtained with Azure Monitoring Tools is shown in the second column of Table 8 and illustrated in Fig. 16.

Taken into account the outcomes of JMeter, in this experiment we can see an average elapsed time of almost 7s, representing a considerable increase of time. The percentile `95%Line` shows a value of almost 15 s. The server load increases but still the rate error is 0%, therefore, all requests have been managed successfully. When analyzing data from Azure Monitoring Tools, we can see an average memory working set of 159.27 MB, which is pretty feasible, and the CPU Time is 17.31 s, which represents the 28.85% of the available time. The average response grows to 2800 ms.

5.2.3. *Experiment #3: Tourists*

For the last load test, we have considered that ENIA will be open to the tourists who visit Andalusia. Tourism in Andalusia is remarkably important. Some typical operations of a tourist profile are regular access to information systems to see components that provide information about beaches water temperature, beaches with access for disabled people, beaches that contain showers as well as components that have any kind of information about points of interest and tourist locations. In 2014, Andalusia received 24 million tourists [73]. If we assume that 2 million tourists visit Andalusia each month, 1% of them uses ENIA mashup application and 10% of them access concurrently to the application, we find that 2000 tourists access to ENIA at the same time. As before, we also assume that each user approximately performs an average of 3 operations in one minute. This means that 6000 requests will be made per minute, that is, 100 requests every second. So, when setting up the JMeter test a new thread group is created and the value of the parameters is set to *number of threads:6000* and *ramp-up period:60*. The results of this experiment can be seen in the third row of Table 7 and graphically illustrated in Fig. 15 marked in brown. Internal data obtained with Azure

Table 8
Azure monitoring tool experiments results.

Metric	30 Requests	1800 Requests	6000 Requests
Average Response Time	413 ms	2800 ms	10970 ms
Average Memory Working Set	32.75 MB	159.27 MB	645.29 MB
CPU Time	0.92 s	17.31 s	49.4 s

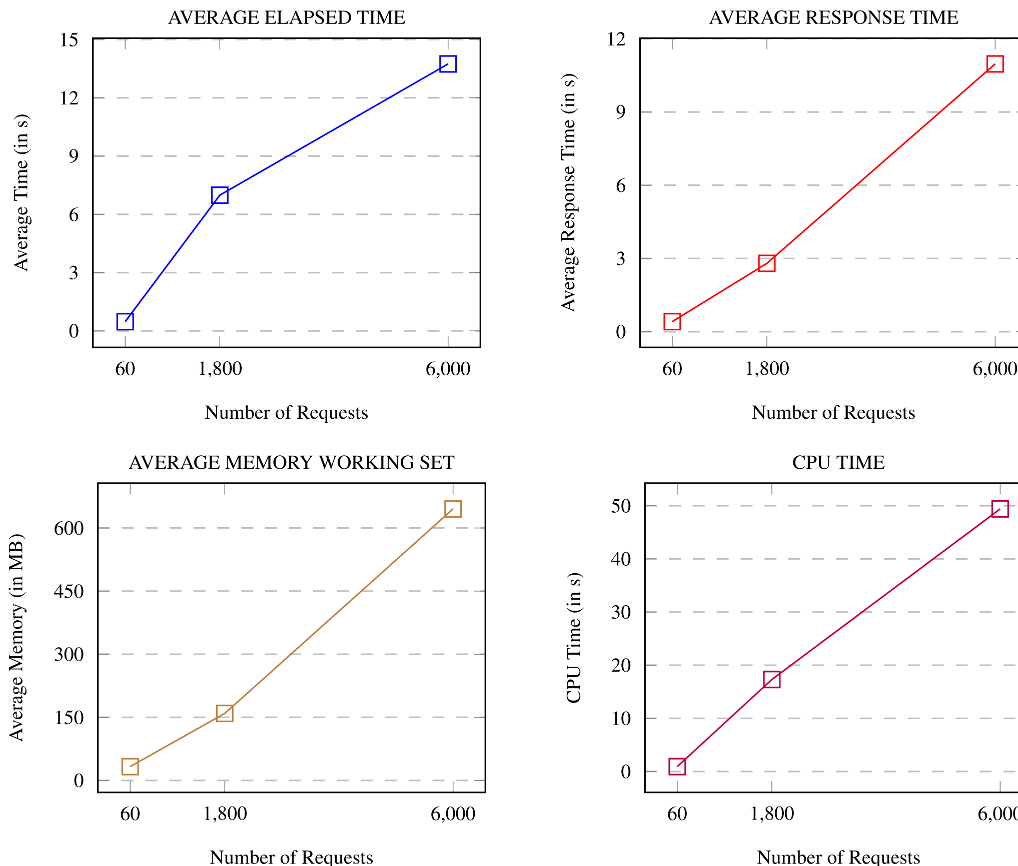


Fig. 16. Azure monitoring tools experiments results.

Monitoring tools can be seen in Table 8 and graphically illustrated in Fig. 16.

The outcomes of Apache JMeter show an average elapsed time close to 14 seconds, which doubles the result of the above experiment. The percentile 95% line shows a value of 30 s.. At this point, the server resources would have been already exceeded if the scalability rules of the cloud resources had not been applied. When analyzing data from Azure Monitoring Tools, we can observe an average memory working set of 645.29 MB that still does not cause any problems and a CPU Time of 49.4 s which represents the 82% of the time available with one instance. As mentioned, the cloud resources have been scaled up adding one more instance for two reasons: The CPU Time surpasses the 80% of the time available and the HTTP queue length surpasses 1000 requests. With this configuration, the data acquisition process can manage the entire income request and the error rate still is 0%.

5.3. Queue management system

As an additional measure, a queue system for managing access to the database has been implemented. As we commented before, the MySQL database is provided by ClearDB as “database-as-a” service. When the data acquisition process manages a request, we cannot guarantee that the interaction will be stored in the database or that the connection to third-party services will be accessible. By implementing that queue we make sure that if we have some problem when dealing with the request because, for example, the database is not responding, we save the interaction in a queue that will be processed further on. We manage that queue as a set of JSON files containing the interaction information as it is received in the HTTP request. Thus, all the interactions received are stored in the database properly. Fig. 17 graphically illustrates the queue.

It is remarkable that the queue system cannot provoke a bottleneck in the system; on the contrary, the queue system can help to recover interactions that could be lost because of an overload or malfunction in the database, third-party web services or due to any other reason. If the cloud resources are capable of processing all the requests, then, the queue system is never used. It is only used when an interaction has not been processed properly. The queue management system works as follows. The JSON information of each interaction that cannot be processed properly is stored as the content of a file in a queue management system folder. There is an automated task that is executed at a certain period of time that processes each file (in that folder that contains information related to an interaction) to save the interaction in the database, then the file is deleted. This task is executed every two minutes but its frequency can be easily modified according to the system needs.

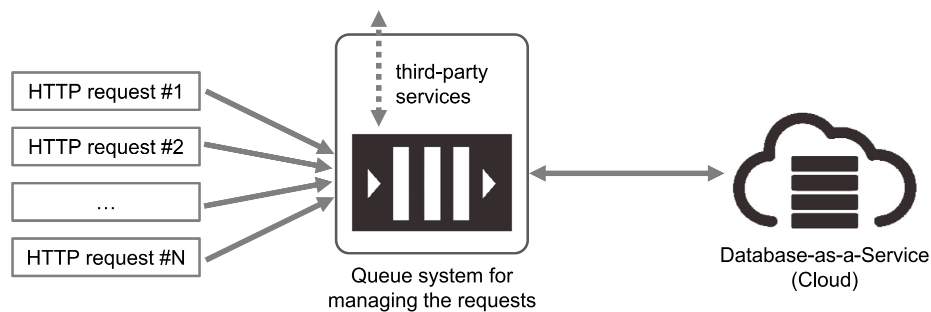


Fig. 17. Queue management for handling HTTP requests.

As commented previously, we implemented our own queue. This was not strictly necessary since there are already many queue management systems. However, most of them require some configuration in servers or cloud resources and we wanted the data acquisition process to be as easy as possible to replicate for other mashup user interfaces. By implementing our own queue, when other developers access the Git repository to clone or create a new brunch, they will not need to deal with servers or cloud configuration or worry about the queue management system, either.

5.4. Experiment conclusion

Once the results of the experiments have been studied, it is possible to conclude that the data acquisition system can withstand the load considering the different situations referred to herein. We might think that the client who sends the request in the third (and harder) experiment has to wait 50 s (the maximum elapsed time found in all experiments), but the truth is that the data acquisition system is totally transparent to the client. In other words, the user's experience when performing an operation in the mashup is not altered with any delay whatsoever. The HTTP request to the data acquisition is made asynchronously from the client without awaiting a reply.

Furthermore, the 50 s delay (the worst time found in the hardest experiment) for storing the interaction in the database is not critical. The purpose of the data acquisition system is to store the interaction for further purposes. For those purposes, accessing to the data and losing an average of 13 s of interactions (the average response time in the hardest experiment performed and worst case scenario possible) does not involve any critical issue and it is simply inconsiderable and feasible.

In summary, an average response time of 0.4, 2.8 and 10.9 s for the REDIAM Staff, Andalusia Staff and Tourist experiments respectively, are acceptable times. It certainly validates the proposed data acquisition process since it is capable of managing the load test with a 100% success rate by storing the interaction.

The data acquisition system is ready to be applied in any mashup applications. Perhaps some mashups have many more users and, consequently, they perform more interactions per minute. Nevertheless, that should not compromise the proper functioning of the data acquisition process. By changing the configuration, the scalability rules and the amount of resources of the cloud pool, the data acquisition process can be scaled up according to the needs. The data acquisition process is located in the cloud as platform-as-a service, just as the database is in the cloud as database as a service. One of the multiple features within cloud computing is scalability [4,49]. Hence, developers should not worry about scalability because the storage provided is virtually infinite [66].

5.5. Mashup comparative performance after the deployment of the data acquisition system

The deployment of the data acquisition system entails an increase in the resources consumption. Fortunately, most of the work is transparent to users and it is managed in the background without altering the user experience. Nevertheless, an experiment has been carried out to evaluate the performance of the ENIA mashup application after the deployment of the data acquisition process to verify that it still have reasonable response times. To contextualize the results of the experiments, we have analyzed the performance of other commercial mashup applications and we have compared the results.

Table 9
Main mashup applications performance comparative.

Operation	ENIA	Geckoboard	CYFE	MyYahoo	Netvibes
Load	4.20 s	3.74 s	2.17 s	7.08 s	1.75 s
Add	0.62 s	4,46 s	1.08 s	1.17 s	1.39 s
Move	0.45 s	0.54 s	0.15 s	0.89 s	0.63 s
Delete	0.87 s	0.91 s	1.67 s	0.55 s	0.07 s

In the previous Section 5.2, we have already validated through the performance metrics that our data acquisition approach to storing the acquired interaction is feasible in terms of computing resources. Now, measuring and analyzing the mashup application response time after the deployment of the data acquisition system we can state whether the formulated working hypothesis can be validated, i.e.- whether it is feasible or not, to create a data acquisition system capable of storing the users' interaction in a transparent way without compromising their user experience.

As analyzed in Section 2 and Section 3, there are some common operations that every mashup has. We have extracted some of these operations (`Add`, `Move` and `Delete`) along the load process of the mashup application to compare the time that takes to perform each one of them. The results are shown in Table 9 and graphically illustrated in Fig. 18.

ENIA mashup application ranks third in the total time value. This position, just in the middle of other mashup applications shows that, even with the data acquisition system running, the response time is consistent with other similar solutions in the market. It is remarkable that when performing the `Add` operation ENIA ranks first, considering that this is one of the operation that can imply further resources consumption because it may need additional requests to the ENIA web services. In any case, ENIA does not rank lower than third when performing any operation used in the comparative.

In conclusion, we can say that our approach to the deployment of a data acquisition system for storing user interactions on mashup user interfaces do not compromise the mashup performance. Accordingly, the objectively measurable working hypothesis defined in the introduction can be answered positively.

5.6. Threats to validity

In order to validate our proposal, we question the possible threats that concern the validity of the data acquisition system. We follow the main types of validity threats used in the literature [25] to assure that

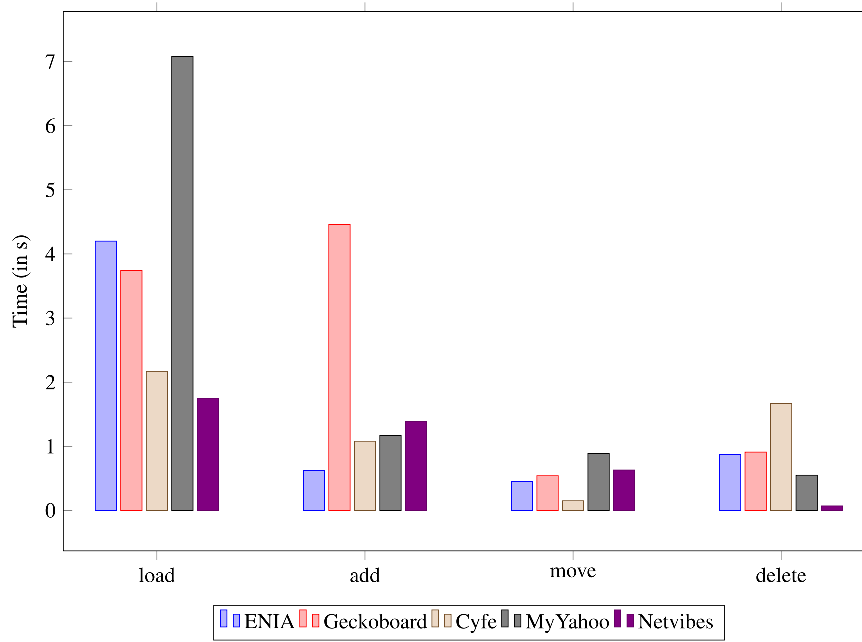


Fig. 18. Mashup applications load time performing operations.

Table 10
Threats to validity.

Validity Type	Question	Answer
Conclusion validity	Does the treatment we introduced have a statistically significant effect on the outcome we measure?	Qualitatively the data acquisition process has a significant effect. After the deployment of the data acquisition system, we store and have at our disposal the users' interaction with mashup application. We can use this data for multiple purposes (data mining, marketing, security, accessibility or usability, among others). Statistically, there is an increase of the resources consumption in the mashup cloud infrastructure. We have studied the impact of the data acquisition system in the mashup performance and the user experience is not diminished because most of the process are transparent to them.
Internal Validity	Did the treatment we introduced cause the effects on the outcome? Can other factors also have had an effect?	The deployment of the data acquisition system does not alter the users' expected outcome. As commented, it is totally transparent to them. The new outcome generated with the user behavior using the mashup is used by internal staff only (developers, analysts, marketing specialist, data scientist...).
Construct validity	Does the treatment correspond to the actual cause we are interested in? Does the outcome correspond to the effect we are interested in?	Our aim is capturing the interaction data and storing it in a relational database. That is exactly what we get after the deployment of the data acquisition system.
External validity	Is the cause and effect relationship we have shown valid in other situations? Can we generalize our results? Do the results apply in other contexts?	The data acquisition can be generalized and deployed in any other mashup application. Depending the technologies used to build the mashup application certain assumptions should be made. In this paper, we deploy the data acquisition system in a real scenario: a component-based architecture using the COScore infrastructure. It has been also applied in other two real scenarios: a mashup that manages home-automation components and a mashup application built using polymer technology from Google.

the relationship between conclusions and reality is not wrong. Table 10 discuss the following validity threats: Conclusion validity, Internal validity, Construct validity and External Validity. These questions are analyzed and by answering positively them we can validate our proposal.

6. Conclusions and future work

In this article, we have presented a data acquisition strategy to allow mashup user interfaces to store the interaction (performed by users over the mashup) into a relational database using a microservice-based architecture. To our knowledge, and in comparison to the above analyzed related work in the literature, the work presented in this article is the first in the *mashup User Interfaces* literature that addresses the problem of capturing the human-computer interactions performed by users over mashup user interfaces through a flexible data acquisition system with the aim of storing such interactions in a relational database.

The article reports the following R&D activities that we have conducted:

— We have deeply investigated the morphology of mashup user interfaces and we have proposed a database schema to store the interaction based, not only on the mashup morphology, but also on the in-

formation managed by these information systems, according to their users and services.
— A microservice architecture has been designed and a data acquisition system has been deployed in order to acquire data from mashup applications whose user interfaces run distributed across multiple devices.
— Each microservice is service-oriented and has its own REST web service to facilitate an agile development as well as an easy software test, inspection and maintenance.
— The interaction data is processed and stored in a relational database so it can be easily accessible for further purposes (e.g., data searching, data mining, marketing, security, accessibility, usability or traceability of interaction data, among others).
— Agile methodologies including Continuous Integration (CI) and SOLID principles have been used in order to ensure the software quality.
— The architecture proposed and, consequently, the data acquisition system have not only been designed but also implemented and deployed in a cloud environment. In this respect, we have defined the cloud infrastructure that sustains the interaction data acquisition system as well as its scalability.
— Descriptions of the key parts of the process as well as several BPMN diagrams have been described. Moreover, a public Git repository with the source code of a real deployment is available.
— Three load tests based on real situations have been performed to analyze the validity of the proposed data acquisition system.
— Both tests and analysis have been applied to a data acquisition system case study in order to validate the deployment and to study its reliability and performance. An empirical study analyzing a real *mashup* User Interface has been discussed.
— A performance comparison of several real mashup applications in the market is done and analyzed.

To validate the study (and the paper), we answer to the research question formulated as working hypothesis by ensuring that:

a) The data acquisition properly extracts the user interaction in mashup application properly;
b) The performance of storing the users interaction in a relational database make our approach affordable;
c) The response time of the data acquisition system is good enough to not harm the user experience in the platform, preserving the mashup application with a response time even better that other solutions in the market, ensuring that the data acquisition system is totally transparent to end-users.

In conclusion, we propose a flexible data acquisition system to acquire and store the users' interaction with mashup user interfaces. We have conducted a performance validation of the data acquisition system deployed in a real scenario obtaining good results and we have discussed the threads of validity that can affect to our proposed strategy with good results. We can state that the proposed approach is feasible and it solves the problem of capturing the users' interaction with mashup software applications. In addition, we provide a guideline that will allow developers to easily adapt this strategy in their applications.

As future work, the proposal presented in this article can be extended and adapted to discover behavioral patterns for creating prediction models by using machine learning, cloud computing and big data techniques alongside software engineering and model transformation perspectives. The exploitation of the kind of data acquired by the data acquisition system proposed in this article can bring some insight that can be inferred to some rules and be applied in mashup user interfaces, leading to a better user's experience.

Regarding to the applications of the algorithms, a considerable amount of data is required. If the collection of data acquired by the data acquisition process in the real case study provided is not enough, a data generator implementation might be needed for further steps. In such case, a semi-arbitrary system could be modeled; this system could be based on real behavioral patterns, identified by user tests which can be carried out by real subjects under the supervision of expertise researchers on this matter.

Finally, as a future challenge, it is a desired goal the evolution of the mashup applications based on the knowledge generated by the machine learning algorithms. Through the automation of data analysis, it might be possible to create a seamless process that could work autonomously towards the evolution of mashup user interfaces.

Acknowledgments

This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Projects TIN2013-41576-R and TIN2017-83964-R. A. J. Fernández-García has been funded bya FPI Grant BES-2014-067974. J. Z. Wang was funded by the US National Science Foundation under Grant No. 1027854. This work was also supported by the CEiA3 and CEIMAR consortia.

References

[1] M. Abrams, C. Phanouriou, A.L. Batongbacal, S.M. Williams, J.E. Shuster, UIML: An appliance-Independent XML user interface language, Comput. Netw. 31 (11–16) (1999) 1695–1708.
[2] C.C. Aggarwal, Mining sensor data streams, in: Managing and Mining Sensor Data, Springer US, 2013, pp. 143–171.
[3] J.M. Almendros-Jiménez, L. Iribarne, An extension of UML for the modeling of wimp user interfaces, J. Visual Lang. Comput. 19 (6) (2008) 695–720, doi:10.1016/j.jvlc.2007.12.004.
[4] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58, doi:10.1145/1721654.1721672.
[5] J.A. Asensio, J. Criado, N. Padilla, L. Iribarne, A safe approach using virtual devices to evaluate home automation architectures prior installations, in: Proceedings of the International Conference on Engineering, Technology and Innovation (ICE/ITMC), 2017, pp. 154–159.
[6] C. Atkinson, T. Kuhne, Model-Driven development: A metamodeling foundation, IEEE Softw. 20 (5) (2003) 36–41, doi:10.1109/MS.2003.1231149.
[7] Azure, Azure Info. https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-sizes-specs, 2016,
[8] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard, The arch model: seeheim revisited (version 1.0), "the UIMS tool developers workshop" (april 1991), ACM SIGCHI Bull. 24 (1) (1992).
[9] S. Berti, F. Correani, G. Mori, F. Paternó, C. Santoro, TERESA: A transformation-based environment for designing and developing multi-device interfaces, in: Proc. of ACM CHI 2004 Conf. on Human Factors in Computing Systems, Vol. II, ACM Press, NY, USA, 2004, pp. 793–794.
[10] F. Bodart, J. Vanderdonckt, On the problem of selecting interaction objects, in: Proceedings of the BCS Conference HCI'94, Cambridge University Press, Glasgow, 1994, pp. 163–178.
[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-oriented software architecture, volume 1: A system of patterns, John Wiley & Sons, 1996.
[12] S. Ceri, P. Fraternali, A. Bongio, Web Modeling Language (WebML): a modeling language for designing web sites, in: Proc. of the 9th Int. WWW Conf. on Computer Networks. Journal of Computer and Telecom. Networking, Amsterdam, The Netherlands, 2000, pp. 137–157, doi:10.1016/S1389-1286(00)00040-2.
[13] C. Chen, C. Zhang, Data-intensive applications, challenges, techniques and technologies: a survey on big data, Inf. Sci. (Ny) 275 (2014) 314–347, doi:10.1016/j.ins.2014.01.015.
[14] M. Chen, S. Mao, Y. Liu, Big data: a survey, Mobile Netw. Appl. 19 (2) (2014) 171–209, doi:10.1007/s11036-013-0489-0.
[15] M. Chinosi, A. Trombetta, BPMN: An introduction to the standard, Comput. Standards Interfaces 34 (1) (2012) 124–134, doi:10.1016/j.csi.2011.06.002.
[16] ClearDB, ClearDB MySQL Databases Characteristics, 2016, (http://w2.cleardb.net/azure/).
[17] CMAYOT, Regional Ministry of Environment and Spatial Planning, Government of Andalusia, Spain (CMAYOT), 2016, (http://www.juntadeandalucia.es/medioambiente).
[18] B. Costa, P.F. Pires, F.C. Delicato, P. Merson, Evaluating REST architectures–approach, tooling and guidelines, J. Syst. Softw. 112 (2016) 156–180, doi:10.1016/j.jss.2015.09.039.
[19] J. Coutaz, Software Architecture Modelling: Bridging Two Worlds Using Ergonomics and Software Properties, Springer, Berlin, Germany, pp. 49–73.
[20] J. Criado, D. Rodríguez-Gracia, L. Iribarne, N. Padilla, Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces, Software 45 (12) (2015) 1677–1718, doi:10.1002/spe.2306.
[21] Cyfe, Cyfe. Business Mashup to manage Social media, analytics, marketing, sales, support and infrastructure components, 2016, (http://www.cyfe.com/).
[22] F. Daniel, M. Matera, Mashups: concepts, models and architectures, Springer International Publishing, 2014.

[23] S. Duarte-Torres, I. Weber, D. Hiemstra, Analysis of search and browsing behavior of young users on the web, ACM Trans. Web 8 (2) (2014) 7:1–7:54, doi:10.1145/2555595.
[24] M. Elkoutbi, R.K. Keller, User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets, in: Proceedings of 21st International Conference on Application and Theory of Petri Nets, ICATPN 2000, in: Lecture Notes in Computer Science, 1825, Springer, Berlin, Germany, 2000, pp. 166–186.
[25] R. Feldt, A. Magazinius, Validity threats in empirical software engineering research - an initial survey. (2010) 374–379.
[26] A. Fernández-García, L. Iribarne, A. Corral, J. Criado, J. Wang, Optimally Storing the User Interaction in Mashup Interfaces within a Relational Database, in: 5th Workshop on Distributed User Interfaces (DUI), LNCS 9881, Springer, 2016, pp. 188–195.
[27] J. Fernandez-Villamor, C. Iglesias, M. Garijo, Microservices - Lightweight Service Descriptions for REST Architectural Style., in: J. Filipe, A.L.N. Fred, B. Sharp (Eds.), ICAART, INSTICC Press, 2010, pp. 576–579.
[28] Geckoboard, Geckoboard. KPI mashup dashboard software for businesses, 2016, (https://www.geckoboard.com/).
[29] X. Guan, B. Cheng, A. Song, H. Wu, Modeling users' behavior for testing the performance of a web map tile service, Transactions in GIS 18 (2014) 109–125, doi:10.1111/tgis.12123.
[30] S.N. Han, I. Khan, G.M. Lee, N. Crespi, R.H. Glitho, Service composition for IP smart object using realtime web protocols: concept and research challenges, Comput. Stand. Interfaces 43 (2016) 79–90, doi:10.1016/j.csi.2015.08.006.
[31] R. Heimgärtner, Identification of the User by Analyzing Human Computer Interaction, Springer Berlin Heidelberg, pp. 275–283. 10.1007/978-3-642-02580-8-30.
[32] V. Hoyer, M. Fischer, Market overview of enterprise mashup tools, in: A. Bouguettaya, I. Krueger, T. Margaria (Eds.), Proceedings of 6th International Conference on Service-Oriented Computing (ICSOC'2008), Springer Berlin Heidelberg, 2008, pp. 708–721, doi:10.1007/978-3-540-89652-4-62.
[33] L. Iribarne, et al.ENIA Project. Environmental Information Agent, 2016, (http://acg.ual.es/projects/enia/).
[34] N. Jan, N. Lin, Web user behaviors prediction system Using trend similarity, in: Proceedings of the 7th WSEAS International Conference on Simulation, Modelling and Optimization, World Scientific and Engineering Academy and Society (WSEAS), 2007, pp. 69–74.
[35] K. Jeffery, D. Kyriazis, A. Ciuffoletti, From distributed to complete computing automated deployment of a microservice-based monitoring infrastructure, Procedia Comput. Sci. 68 (2015) 163–172, doi:10.1016/j.procs.2015.09.232.
[36] JMeter, Apache JMeter. Load test and measure performance tool, 2016, (http://jmeter.apache.org/).
[37] K. Kinley, D. Tjondronegoro, H. Partridge, S. Edwards, Modeling users' web search behavior and their cognitive styles, J. Assoc. Inf. Sci. Technol. 65 (6) (2014) 1107–1123, doi:10.1002/asi.23053.
[38] P. Lake, R. Drake, Information systems management in the big data era, Advanced Information and Knowledge Processing, Springer International Publishing, 2014.
[39] S. LaValle, E. Lesser, R. Shockley, M. Hopkins, N. Kruschwitz, Big data, analytics and the path from insights to value, MIT Sloan Management Review 52 (2) (2011) 21–32.
[40] K. Leung, L.C.K. Hui, S. Yiu, R. Tang, Modeling web navigation by statechart, in: Proceedings of the 24th Annual International Computer Software and Applications Conference, COMPSAC'2000, IEEE Computer Society Press, Washington, DC, USA, 2000, pp. 41–47.
[41] J. Lewis, M. Fowler, Microservices: a definition of this new architectural term, 2014, (http://martinfowler.com/articles/microservices.html).
[42] F. Liang, H. Guo, S. Yi, S. Ma, A scalable data acquisition architecture in web-based iot, in: Int. Conf. on Information and Business Intelligence (IBI 2011), Chongqing, China, December 23–25, Springer Berlin Heidelberg, 2012, pp. 102–108, doi:10.1007/978-3-642-29084-8-16.
[43] B. Lieberman, UML activity diagrams: Detailing user interface navigation, Technical Report, https://www.ibm.com/developerworks/rational/library/4697.html, 2001.
[44] Y. Liu, X. Liang, L. Xu, M. Staples, L. Zhu, Composing enterprise mashup components and services using architecture integration patterns, J,. Syst. Softw. 84 (9) (2011) 1436–1446, doi:10.1016/j.jss.2011.01.030.
[45] D. Lizcano, F. Alonso, J. Soriano, G. López, A component- and connector-based approach for end-user composite web applications development, J. Syst. Softw. 94 (2014) 108–128, doi:10.1016/j.jss.2014.03.039.
[46] D. Lizcano, G. López, J. Soriano, J. Lloret, Implementation of end-user development success factors in mashup development environments, Comput. Standards Interfaces 47 (2016) 1–18, doi:10.1016/j.csi.2016.02.006.
[47] R.K. Lomotey, R. Deters, Architectural designs from mobile cloud computing to ubiquitous cloud computing - survey, in: 2014 IEEE World Congress on Services, 2014, pp. 418–425, doi:10.1109/SERVICES.2014.79.
[48] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, A. Ghalsasi, Cloud computing - the business perspective, Decis. Support Syst. 51 (1) (2011) 176–189, doi:10.1016/j.dss.2010.12.006.
[49] P.M. Mell, T. Grance, SP 800-145. The NIST Definition of Cloud Computing, Technical Report, 2011. Gaithersburg, MD, United States.
[50] G. Mesfin, T.-M. Grønli, D. Midekso, G. Ghinea, Towards end-user development of REST client applications on smartphones, Comput. Standards Interfaces 44 (2016) 205–219, doi:10.1016/j.csi.2015.08.004.
[51] J. Molina, P. González, M. Lozano, Human-Computer Interaction. Theory and Practice, Lawrence Erlbaum Associates.
[52] G. Mori, F. Paternò, C. Santoro, CTTE: Support for developing and analyzing task models for interactive system design, IEEE Trans. Software Eng. (2002) 797–813.
[53] M. Muehlen, J. Nickerson, D. Midekso, G. Ghinea, Developing web services choreography standards–the case of REST vs. SOAP, Decis. Support Syst. 40 (1) (2005) 9–29, doi:10.1016/j.dss.2004.04.008.
[54] B. Myers, S.E. Hudson, R. Pausch, Past, present, and future of user interface software tools, ACM Trans. Comput. Human Interact. 7 (1) (2000) 3–28, doi:10.1145/344949.344959.
[55] MyYahoo, Customizable web mashup with news, stock quotes, weather, and many other features, 2016, (https://my.yahoo.com/).
[56] Netvibes, Mashup application for analyze and act on all the data that matters to a brand or business., 2016, (https://www.netvibes.com/).
[57] A. Newell, S. Card, The prospects for psycological science in human-Computer interaction, Human-Comput. Interact. 1 (1985) 209–242.
[58] N.J. Nunes, Representing User-Interface Patterns in UML, in: Proceedings of the International Conference on Object Oriented Information Systems, OOIS' 2003, in: Lecture Notes in Computer Science, 2817, Springer, Berlin, Germany, 2003, pp. 142–151.
[59] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, Comparison of JSON and XML Data Interchange Formats: A Case Study, CAINE, 2009, pp. 157–162.
[60] S.A.M. Offermans, H.A. van Essen, J.H. Eggen, User interaction with everyday lighting systems, Pers Ubiquitous Comput. 18 (8) (2014) 2035–2055, doi:10.1007/s00779-014-0759-2.
[61] OGC, The Open Geospatial Consortium (OGC), 2016, (http://www.opengeospatial.org/).
[62] C. Pautasso, E. Wilde, R. Alarcon, REST: Advanced research topics and practical applications, Springer New York, 2014.
[63] P. Pinheiro da Silva, N.W. Paton, User Interface Modeling in UMLi, IEEE Software 20 (4) (2003) 62–69.
[64] A. Puerta, J. Eisenstein, XIML: A Common Representation for Interaction Data, in: Proceedings of the 7th International Conference on Intelligent user interfaces, IUI '02, ACM Press, NY, USA, 2002, pp. 214–215, doi:10.1145/502716.502763.
[65] REDIAM, Andalusia Environmental Inf. Network, 2016, (http://www.juntadeandalucia.es/medioambiente/site/rediam/).
[66] S. Sakr, A. Liu, D.M. Batista, M. Alomari, A survey of large scale data management approaches in cloud environments, IEEE Commun. Surv. Tutorials 13 (3) (2011) 311–336, doi:10.1109/SURV.2011.032211.00087.
[67] S. Sathe, T.G. Papaioannou, H. Jeung, K. Aberer, A Survey of Model-based Sensor Data Acquisition and Management, Springer US, Boston, MA, pp. 9–50. 10.1007/978-1-4614-6309-2-2.
[68] S. Scherer, M. Glodek, G. Layher, M. Schels, M. Schmidt, T. Brosch, S. Tschechne, F. Schwenker, H. Neumann, G. Palm, A generic framework for the inference of user states in human computer interaction, J. Multimodal User Interfaces 6 (3) (2012) 117–141, doi:10.1007/s12193-012-0093-9.
[69] SOAP, SOAP (Simple Object Access Protocol) W3C Standard, 2007, (https://www.w3.org/TR/soap12/).
[70] A. Stanciulescu, Q. Limbourg, J. Vanderdonckt, B. Michotte, F. Montero, A transformational approach for multimodal web user interfaces based on UsiXML, in: Proc. of the 7th Int. Conf. on Multimodal Interfaces, ICMI '05, ACM Press, NY, USA, 2005, pp. 259–266, doi:10.1145/1088463.1088508.
[71] M. Swobodzinski, P. Jankowski, Evaluating user interaction with a web-based group decision support system: a comparison between two clustering methods, Decis. Support Syst. 77 (2015) 148–157, doi:10.1016/j.dss.2015.07.001.
[72] H. Trætteberg, UI Design without a Task Modeling Language – Using BPMN and Diamodl for Task Modeling and Dialog Design, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 110–117.
[73] Turismoandaluz, Tourist Statistics and Analysis in Andalucia, 2016, (http://www.turismoandaluz.com/estadisticas/).
[74] J. Vallecillos, J. Criado, A.J. Fernández-García, N. Padilla, L. Iribarne, Service-oriented computing – ICSOC 2015, WESOA Workshop, Springer Berlin Heidelberg, pp. 64–75. 10.1007/978-3-662-50539-7-6.
[75] J. Vallecillos, J. Criado, N. Padilla, L. Iribarne, A cloud service for COTS component-based architectures, Comput. Standards Interfaces 48 (2016) 198–216, doi:10.1016/j.csi.2015.11.008.
[76] J. Vanderdonckt, F. Bodart, Encapsulating knowledge for intelligent automatic interaction objects selection, in: Proceedings of the Conference on Human Factors in Computing Systems, INTERCHI'93, ACM Press, NY, USA, 1993, pp. 424–429.
[77] M. Winckler, P. Palanque, StateWebCharts: A formal description technique dedicated to navigation modelling of web applications, in: Proc. of the 10th Int. Workshop on Interactive Systems. Design, Specification, and Verification, DSV-IS 2003, in: Lecture Notes in Computer Science, 2844, Springer, Berlin, Germany, 2003, pp. 61–76.
[78] S. Youssefi, S. Denei, F. Mastrogiovanni, C. G., Skinware: A real-time middleware for acquisition of tactile data from large scale robotic skins, in: 2014 IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 6421–6426, doi:10.1109/ICRA.2014.6907807.
[79] F. Zhang, X. Li, Research in automatic search engine replacement algorithm for web caching based on user behavior, in: 7th Web Information Systems and Applications Conference (WISA), 2010, pp. 142–145, doi:10.1109/WISA.2010.25.
[80] K. Zheng, R. Padman, M. Johnson, H. Diamond, An interface-driven analysis of user interactions with an electronic health records system, in: Journal of the American Medical Informatics Association, 2009, pp. 228–237, doi:10.1197/jamia.M2852.