Open Linked Widgets Mashup Platform

Tuan-Dat Trinh[1], Peter Wetz[1], Ba-Lam Do[1], Amin Anjomshoaa[1]
Elmar Kiesling[1], A Min Tjoa[1]

[1]Vienna University of Technology, Vienna, Austria
{tuan.trinh, peter.wetz, ba.do, anjomshoaa
elmar.kiesling, amin}@tuwien.ac.at

Abstract. Since the emergence of the mashup concept on the web around 2005, a large stream of academic research and industrial development resulted in numerous architecture proposals, platforms and editing tools. This strong initial interest in mashup technologies and promising use case demonstrations notwithstanding, however, commercial platforms such as Microsoft Popfly, IBM Mashup Center, and Google Mashup Editor failed to gain widespread adoption by consumers and enterprises and were eventually discontinued. This failure may be attributed to a number of common limitations of these platforms: (i) they are each useful only for a single or a limited number of restricted problems in specific domains; (ii) they are closed, i.e., developers cannot contribute and share their widgets; (iii) widgets, which are crucial elements of any mashup platform, are usually not modeled in sufficient semantic detail to support widget search and composition features that facilitate reuse. This paper addresses these limitations by introducing an open mashup platform based on semantic web technologies. We present a novel architecture in which widgets equipped with a semantic, graph-based model can cooperate with each other in a mashup created by end users through simple drag and drop operations. Widgets created freely by independent developers and hosted on arbitrary servers can be discovered and combined easily through our introduced semantic search feature.

1 Introduction

Mashups are based on the idea of *"using content from more than one source to create a single new service displayed in a single graphical interface*" [5], thereby making existing data more useful. Mashup platforms aim to facilitate the effortless creation of such mashups in an ad-hoc manner in order to allow users to solve a problem at hand, e.g., a specific need for information without the need to acquire programming skills.

After commercial mashup development environments such as *Microsoft Popfly*, *Yahoo Pipes*, or *IBM Mashup Center* were launched in 2006, both mashup research and mashup tool development progressed rapidly. However, most of these tools - with few exceptions such as *Yahoo Pipes*[1] and *Presto*[2] - are no longer available today.

[1] http://pipes.yahoo.com/pipes/

Even though mashup ideas attracted significant academic and public interest initially, end users did not widely adopt the available mashup platforms. Surveys of the mashup literature [1, 2] have developed a number of evaluation criteria and identified shortcomings of existing approaches. Some of these shortcomings have been addressed in more recent contributions, but others remain an open challenge. We address the following three crucial limitations: (i) current platforms are too domain-specific and are applicable only for a small set of limited problems; (ii) the input and output data paths of widgets are not modeled clearly, which makes it hard for users to quickly and effortlessly create a mashup when the number of widgets is large; (iii) widgets, i.e., the fundamental elements of any mashup platform, are restricted in the functionality they provide and cannot be extended externally by the community. Consequently, users need to adopt and use multiple different mashup platforms for various purposes, which results in a dissatisfaction mashup experience. Although we cannot directly combine the advantages of different platforms because of the heterogeneous widget data models and architectures, it is possible to modify widgets of a platform so that they can operate in another one. To this end, it is necessary that the target platform is truly open. *IBM Mashup Center*[3], for example, is a relatively open platform since its widgets can be written in any language by developers. However, it requires each enterprise or organization to deploy the platform on their own servers with their own proprietary widgets. Furthermore, it is a commercial product that has been discontinued.

To address these issues, this paper presents an open mashup platform based on semantic web technologies. In this platform, a widget does not only solve a particular problem for the developer who creates it, but may also serve as a reusable building block that can be applied by other users for solving larger problems in future. This approach does not neglect the importance of domain-specific mashups. Rather, users can still implement arbitrary widgets for particular domains, and widgets from different domains can be combined to solve cross-domain problems.

We introduce a *Linked Widget Model*, i.e., a framework for developing such widgets and an architecture for their cooperation. A key advantage of our approach is that widgets exchange data in *JSON-LD*[4] format, which can be easily converted into *RDF*. This adds a semantic layer to the data and makes it machine-readable. Furthermore, this allows the platform to present sample data for the models in *JSON-LD* format as soon as developers have properly annotated a widget.

In developing this model, we address the following research challenges: (i) How can users be supported in selecting and combining appropriate widgets for their mashups?; and (ii) How can widgets cooperate in a mashup even though they are unaware of each others' existence? We tackle the first research question using a *semantic web* approach outlined in Section 3. The later question will be addressed in Section 5, which presents the underlying messaging protocol.

---

[2] http://mdc.jackbe.com/enterprise-mashup

[3] http://pic.dhe.ibm.com/infocenter/mashhelp/v3/index.jsp

[4] http://www.w3.org/TR/json-ld/

We illustrate the resulting platform via a prototypical web application.[5] This open platform is targeted at both developers and end users. Developers can implement widgets in any client/server web programming language (e.g., *HTML*, *JS*, *PHP*, *JSP* or *ASP)* and contribute them to the platform. In particular, widgets can be deployed on arbitrary servers outside the platform, which makes maintenance economical and scalable. Each widget and each mashup has its own URIs, which can be used for sharing or publishing purposes. Anyone can create a mashup via this platform and publish it on their personal websites and others may freely modify and republish them on their own websites. In this context, the semantic description model makes the data more meaningful and facilitates sharing and reusability.

The remainder of this paper is organized as follows. In Section 2, we introduce key terms and outline the architecture of our proposed platform. Section 3 introduces example use cases and the linked widget model; Section 4 illustrates the widget development process; Section 5 is dedicated to the cooperation between widgets. The paper concludes in Section 6, providing an outlook on future research.

2 Terms and Architecture

Before presenting the platform architecture, we define a set of basic terms and concepts. A *Widget*, a basic component of a mashup, is an "*interactive single purpose application for displaying and/or updating local data or data on the Web, packaged in a way to allow a single download and installation on a user's machine or mobile device*"[6]. Such widgets have the potential to use the available web services and web APIs and also access the existing open data such as Open Governmental Data (OGD) and Linked Data. We introduce *Linked Widgets* [3] as the key concept of our open mashup platform. Linked Widgets are standard widgets enhanced with a semantic model following the Linked Data principles. The semantic model describes the data input/output and other information such as provenance, license, etc. In particular, a *Linked Widget* consists of four main components: (i) *input terminals*, (ii) *output terminals*, (iii) *options*, and (iv) a *processing function*. Input/output terminals are used to connect widgets in a mashup and represent the data flow. Options are *HTML* inputs inside a widget. They provide a mechanism that allows users to control widgets behavior. Finally, the processing function defines how widgets receive input and return their output.

We distinguish four basic types of widgets: *data widget*, *processing widget*, *presentation widget*, and *user interaction widget*. Each widget can have more than one type. A *data widget* retrieves data from a data source and provides the collected data for others. Hence, it has no input terminals. A *processing widget* takes input data from other widgets and handles the data before returning the results. It therefore has both input and output terminals. A *presentation widget* has at least one input so that it can receive and represent data from another widget visually.

[5] http://linkedwidgets.org
[6] http://www.w3.org/TR/widgets/

As an example, consider a mashup with three widgets $A \rightarrow B \rightarrow C$. Typically, when the user triggers an action to run a Widget, e.g., C, this action requires all widgets that provide input to this widget, i.e. A, and B, to run first since C needs the output from B and B, in turn, need the output from A. Inversely, when an event is fired at A, e.g., mouse-clicked events, result of B and C must also be updated. In the latter case, A is a *user interaction widget* for which developers can define user actions, e.g. item selection, as a trigger to update all widgets lying behind it in a mashup.

Mashup is the combination and interconnection of widgets. A mashup should contain at least one data widget providing the data and one presentation widget to show the final results.

Fig. 1 illustrates the architecture of the platform. Developers implement and deploy their widgets on an arbitrary server. Then the widget URL is passed to the *widget annotator module*, which adds semantic annotations. After the validation process, the created widget model is stored permanently as linked data that can be accessed through a SPARQL endpoint. The URIs can be dereferenced through tools such as *Pubby*.[7]

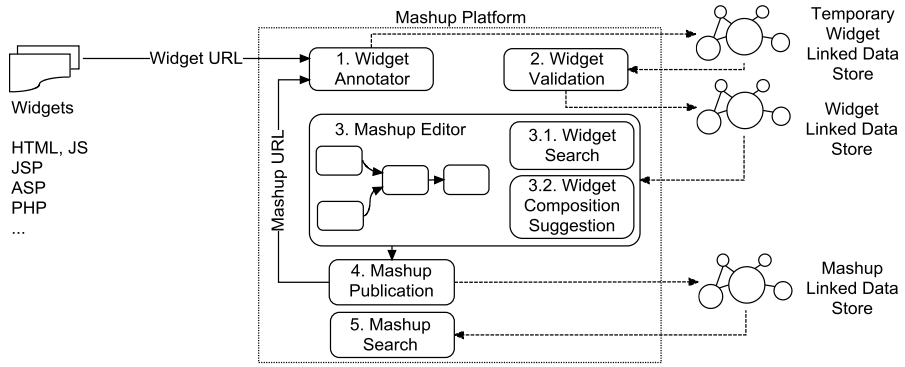

Fig. 1. Linked Widgets platform architecture

The first step in creating a mashup solution is to search for available widgets. Any terminal of a widget can be selected and queried to find compatible terminals from other widgets. The list of compatible widgets will then be presented to users in order to support them during the mashup creation. This is achieved by executing a *SPARQL* query over the semantic data repository of widget models.

The final result of the mashup is directly displayed on the platform. Alternatively, it can also be displayed via the *mashup publication module* and be shared and published on other websites. The mashup itself can also be saved as a new widget using this module. This encourages users' creativity by allowing them to create mashups and reusing them in other mashups without programming skills.

In conclusion, the platform architecture is designed in a dynamic, flexible, and scalable manner. It is open and can be extended with an arbitrary number of highly versatile widgets. Maintaining this platform is economical since widgets can be stored

[7] http://wifo5-03.informatik.uni-mannheim.de/pubby/

externally and both the data retrieval and data processing tasks take place in either the client's browser or on the widget server.

3 Linked Widget Model

Before being made accessible to users, a widget needs to be semantically annotated by developers and validated by the platform manager. To this end, we enrich the widget's input and output data with semantic models. These semantic I/O models are essential for the subsequent search and composition processes. Furthermore, they are crucial for effective sharing of widgets as the usefulness of the platform would deteriorate quickly as the number of widgets in the platform increases without appropriate semantic filtering mechanisms. For example, even when the number of widgets available is fairly limited (e.g. 43 for *Yahoo Pipes* and 300+ for *Microsoft Popfly*), finding the appropriate widgets needed for building a mashup solution is already a difficult task. Existing mashup platforms usually employ a text-based approach for widget search, which is not very helpful for advanced widget exploration and widget composition tasks.

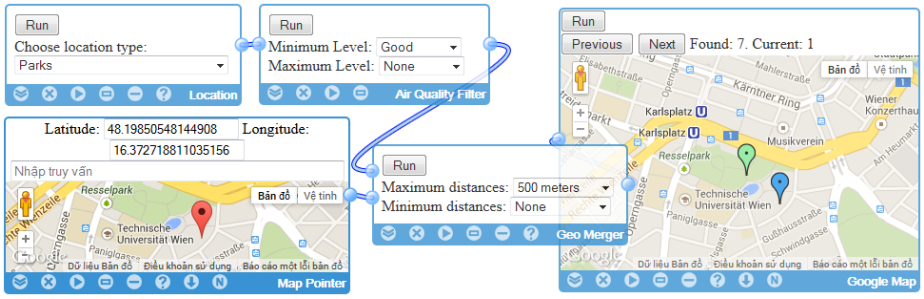

Fig. 2. A mashup example[8]

To deal with a potentially large number of widgets in future, we organize widgets in a taxonomy and *widget collections*. Developers or end users create a widget collection by collecting a list of widgets capable of solving a certain class of problems, e.g. a *Vienna point of interest (POI)* collection, a *Berlin POI* collection or a *Flickr* collection. Such collections make it much easier for other users to select appropriate widgets for their mashup. Consider, for example, that a *Vienna POI* collection might include five widgets: (1) *Google Map* – displays points on the map, (2) *Location* – returns list of POI, e.g. Park, Hospital, (3) *Air Quality* – calculates and filters air quality for points, (4) *Geo Merger* – returns nearby points from two arrays of points, (5) *Map Pointer* – creates a point on Google map. By combining these widgets in a mashup, the collection can answer lots of different questions such as "*find all parks in Vienna*", "*find all parks near some swimming pool*", "*find a combination of park, swimming pool, museum near each other*", "*find a park with good air quality and not more*

[8] More examples are available on http://linkedwidgets.org

than 500m far from a point in the map", etc. The mashup for the last question is depicted in Fig. 2.

Fig. 3 presents a part of our ontology for the semantic modeling of Linked Widgets. It also shows the detailed model of *Geo Merger*. The widget receives two arrays of arbitrary objects containing the *wgs84:location* property. Its domain is the *Point* class with two literal properties, i.e. *lat* and *long*. The widget output is a two-dimensional array in which each row represents two objects from two input arrays, respectively. Those objects have locations satisfying the distance filter of the *Geo Merger* widget.

The input and output of *Map Pointer*, *Google Map* and *Air Quality Filter* can be modeled in a similar way. To represent their data models as an "arbitrary" object, we utilize the *owl:Thing* class. The data model of the *Location* widget is an instance of the *Place* class with *name*, *address* and *location* properties.

To specify that this input is an array of objects, the literal property *hasArrayDimension* is used (0: single element; n ≥ 1: n multi-dimensional array). In the example, we need *thing2* and cannot use *thing1* as the data model of *output1* because the output's dimension is 2. The *Point*, *Place*, *location*, *lat* and *long* resources here can be obtained from any ontology. However, since a well-established ontology facilitates the data exchange process between widgets, *wgs84* was chosen. Generally, we collect and recommend the most popular resources to developers in the model creation process.

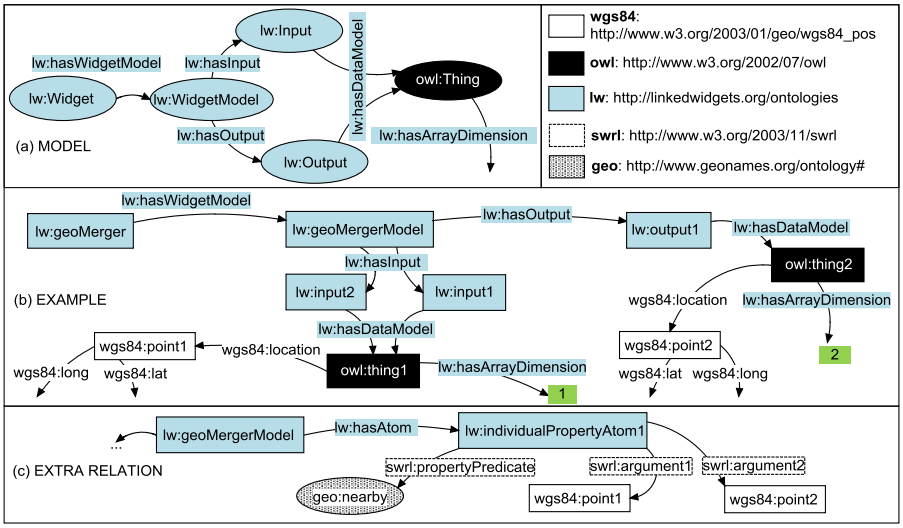

Fig. 3. General Linked Widget model and *Geo merger* model

We have developed a GUI annotator tool that allows developers to rapidly create the complex Linked Widget models that are crucial for widget search and composition features. With *SPARQL* queries, we, for example, can find a widget which receives/outputs an object containing geo information. Furthermore, from an input terminal, e.g. the first input of *Geo Merger*, we can find all output terminals that can be

connected with it as the query shown in Fig. 4. The conditions for these terminals are that they have the same type, the same array dimension and that their set of attributes is a subset of that of the output terminal. In the *Vienna POI* collection, the outputs of *Location*, *Map Pointer* and *Air Quality* satisfy this query.

Similarly, we can model a more complex widget. Its input and/or output have object attributes and there can be relations between those objects. For example, when modeling the *Geo Merger*, if required, we can present the *nearby* relation between *point1* and *point2* as shown in **Fig. 3**.

Using semantic web technologies to describe mashups and their components is not a new approach (cf. [6,7]). However, rather than following a service-based approach like existing platforms, we use a graph-based approach [8, 9, 10] to clearly annotate the input, output, their components and their relations. As the result, the graph-based description can answer more complex questions such as "*find all widgets containing the nearby relation between two locations*".

```
PREFIX ifs: <http://ifs.tuwien.ac.at/>
SELECT DISTINCT ?oTerminalName ?oWidget WHERE {
  <http://ifs.tuwien.ac.at/WidgetGeoMerge>  ifs:hasWidgetModel  ?iWModel.
  ?iWModel ifs:hasInput  [ifs:hasName "input1"^^xsd:string;
                          ifs:hasDataModel ?iDataModel].
  ?iDataModel a ?type.
  ?iDataModel ifs:hasArrayDimension ?listLevel.

  ?oWidget ifs:hasWidgetModel  ?oWModel.
  ?oWModel  ifs:hasOutput [ifs:hasName ?oTerminalName;
                           ifs:hasDataModel ?oDataModel]
  ?oDataModel a [rdfs:subClassOf ?type].
  ?oDataModel ifs:hasArrayDimension ?listLevel.

  FILTER NOT EXISTS{
    ?iDataModel ?property ?iValue.
    FILTER NOT EXISTS {?oDataModel ?property ?oValue.}
  }
}
```

Fig. 4. A SPARQL query for terminal matching

4 Widget Development

Widgets can be developed either through programming, or by end users without programming skills by creating a mashup and saving it as a widget.

The data format for input/output is *JSON-LD*. There are three steps to program a widget: (i) inject a JS file from the platform into the widget to equip it with the capability of cooperating with others; (ii) define input/output configuration; (iii) implement the *JS run(data)* function defining the way the widget processes input data. The *data* object here is *null* if it has no input terminal. Otherwise, during runtime, the platform collects data from all output terminals which connect with the widget's input terminals to build the *data* object and pass it as a parameter for the *run* function. An efficient method for creating a widget is annotating the widget first so that the

platform can automatically generate the widget (and its sample input/output data in form of *JSON-LD*). Developers can then implement the *processing* function.

5 Widget Cooperation

Widgets in a mashup collaborate with each other to form a processing flow, which can be either a *normal flow* or a *user interaction flow*. In a *normal flow*, if a widget needs to run, it first requests its input. Since this input is connected with the output of other widgets, those widgets need to run, too. This process continues until the related widgets have no input. Data are transferred between widgets in a successive process.

Because widgets do not know of each other, they have to communicate with the platform to obtain their jobs. Fig. 5 shows all messages transferred between the platform and widgets for the use case presented in Fig. 2. The first two messages are delivered when widgets are created for the mashup. The platform sets IDs for all widgets and then receives their terminal configurations. Remaining messages are used when *Google Map* runs, in a *normal flow*. They are all transparent to developers.

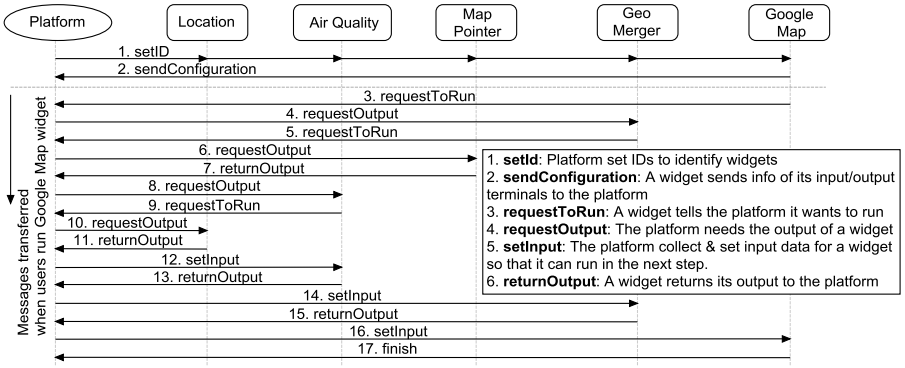

Fig. 5. An example of widget communication

A *User interaction flow* is similar to a *normal flow*, but with reversed direction. The mashup must contain a user interaction widget, e.g. the *Map Pointer* in our example. When an event is fired, e.g. users create a Point by clicking on the map, the final result presented in *Geo Merger* needs to be updated. If *Air Quality* has run before and already has generated output, it is not necessary for it to run again. Only *Geo Merger* and *Google Map* are required to rerun to update the final result.

The communication is taking place purely in the client's browser since a *Window* can send/receive messages to/from others. After the mashup has been created, the platform server is no longer needed since the browser and the widget's servers compute the jobs. This process reduces the platform-server load and improves performance and scalability. For big data and slow client devices, we are developing algorithms to execute every job on the server side and return only the final result to end users.

6 Conclusion and Future Work

This paper presents an overview of an open mashup platform that is scalable, generic, and sharable. It is also simple enough to be used effectively by end users, who have no knowledge about semantic web technologies or programming. Because there are no restrictions on the number of widgets, widget developers and their chosen languages, the platform can solve various problems. It is the first mashup platform that utilizes *JSON-LD* to exchange linked data between widgets. With the Linked Widget model, the inputs and outputs are clearly modeled, enabling users to combine and search widgets based on defined semantics rather than a text-based only search approach.

Since this platform is still in its early stages, we want to shortly discuss the larger vision and direction we are pursuing. In future, this system should serve as a **data platform for the people**. The Open Linked Widgets Platform shall bring together both mashup developers and mashup users. For each of them it should be as easy as possible to create, use, reuse, modify and execute available or newly created mashups. As a consequence, citizens will be enabled to work with all different kinds of data sources, e.g. government data, financial data, environmental data, and data types, e.g. open data, linked data, tabular data, without having to worry about any related barriers. On top of that, new knowledge can be deduced and created by enabling creative (re)combination of different data. Therefore, from a data perspective, the vision is to provide a means to support people in their everyday decision-making.

From a more technical perspective, we aim at providing a **best practice semantic web platform** in the area of mashups. The goal is to fulfill the semantic web vision by implementing the system based on the well-known and defined semantic web principles. We also want to push these ideas one step further by transferring non-semantic data on a semantic level and provide it via our platform in an automated manner. This approach shall enable full utilization of the data's potential.

Future research will focus on using the semantic models, especially for the widget composition feature. The model-matching algorithm also needs to be improved by utilizing ontology alignment techniques, since two models can use different resources from different ontologies. Another interesting field is the automatic creation of new widgets, which are able to handle dynamic web data as an input source.

References

1. Aghaee, S., Pautasso, C.: An Evaluation of Mashup Tools Based on Support for Heterogeneous Mashup Components. In: Harth, A. and Koch, N. (eds.) Current Trends in Web Engineering. Springer Berlin Heidelberg (2012) 1–12
2. Grammel, L., Storey, M.-A.: A Survey of Mashup Development Environments. In: Chignell, M. et al. (eds.) The Smart Internet. Springer Berlin Heidelberg (2010) 137–151
3. Trinh, T.D. et al.: Linked Widgets-An Approach to Exploit Open Government Data. Proceedings of the 15th International Conference on Information Integration and Web-based Applications & Services. ACM (2013) 438–442

4. Yu, J. et al.: Understanding Mashup Development. IEEE Internet Computing vol. 12, issue 5 (2008) 44-52
5. Nicole, C.E., Jenny, L.: Library Mashups-Exploring New Ways to Deliver Library Data (2009)
6. Pietschmann, S., Radeck, C., Meißner, K.: Semantics-based discovery, selection and mediation for presentation-oriented mashups. Proceedings of the 5th International Workshop on Web APIs and Service Mashups. ACM, New York (2011) 7:1–7:8
7. Ngu, A. H. H., Carlson, M. P., Sheng, Q. Z., Paik, H.-y.: Semantic based mashup of composite applications. IEEE Transactions on Services Computing, vol. 3, no. 1 (2010) 2–15
8. Mohsen, T., Craig, A. K., Pedro, S., Jose L. A.: Rapidly Integrating Services into the Linked Data Cloud. Lecture Notes in Computer Science, Vol. 7649. Springer Berlin Heidelberg (2012) 559–574
9. Mohsen, T., Craig, A. K., Pedro, S., Jose L. A.: A Graph-Based Approach to Learn Semantic Descriptions of Data Sources. Lecture Notes in Computer Science, Vol. 8218. Springer Berlin Heidelberg (2013) 607–623
10. Verborgh, R.; Steiner, T.; Deursen, D.V.; Van de Walle, R.; Valles, J.G.: Efficient runtime service discovery and consumption with hyperlinked RESTdesc. Proceedings of the 7th International Conference on Next Generation Web Services Practices (NWeSP) (2011) 373–379