A Component-Based Approach for Engineering Enterprise Mashups

Javier López[1], Fernando Bellas[1], Alberto Pan[1], and Paula Montoto[2]

[1] Information and Communications Technology Department, University of A Coruña
Facultad de Informática, Campus de Elviña, s/n, 15071, A Coruña, Spain
{jmato,fbellas,apan}@udc.es
[2] Denodo Technologies, Inc.
Real 22, 3º, 15003, A Coruña, Spain
pmontoto@denodo.com

Abstract. Mashup applications combine pieces of functionality from several existing, heterogeneous sources to provide new integrated functionality. This paper presents the design of an enterprise-oriented mashup tool that fully supports the construction of mashup applications. Our tool provides generic, reusable components to engineer mashup applications. It includes components for accessing heterogeneous sources, a component to combine data from different sources and components for building the graphical interface. The user builds graphically the mashup application by selecting, customizing, and interconnecting components. Unlike other proposals we: (1) use the Data Federation/Mediation pattern (instead of Pipes and Filters pattern) to express the data combination logic, (2) follow the RESTful architectural style to improve component reusability, and (3) reuse Java standard portal technology to implement the graphical interface of the mashup application.

Keywords: Enterprise Mashups, Integration Patterns, Web Engineering.

1 Introduction

Mashup applications combine pieces of functionality from several existing, heterogeneous sources to provide new integrated functionality. Many research works and industrial tools have appeared during the last two years to ease the creation of mashups. In particular, mashup tools enable 'power users', which do not have programming skills, to build easily mashup applications to quickly respond to a personal or business need.

There are two key aspects in building a mashup application: (1) accessing data sources to obtain unified data and (2) building the graphical interface. Some tools are strong in the first aspect (e.g. [23][13][19]), and only include minimum graphical support to format the returned data. Other tools (e.g. [9][10][4]) support both aspects. Tools also differ in the type of mashup application they are oriented to. Some of them are hosted tools (e.g. [23][13]) that allow Internet users to build mashup applications ('consumer' mashups). Other tools (e.g. [9][10]) are enterprise-oriented and allow the

M. Gaedke, M. Grossniklaus, and O. Díaz (Eds.): ICWE 2009, LNCS 5648, pp. 30 – 44, 2009.

construction of 'enterprise' mashups. Consumer-oriented tools include support to integrate a limited set of sources (typically, RSS/Atom feeds, and REST services) and provide basic graphical capabilities. Enterprise-oriented tools need to take into account many other types of sources, such as databases, SOAP services, and semi-structured HTML web sources, or even to extract parts of web pages (web clipping). Furthermore, the data combination logic and the requirements of the graphical interface are often much more complex.

In this paper we present the design of an enterprise-oriented mashup tool that fully supports the construction of mashup applications. Our tool provides generic, reusable components to engineer mashup applications. It includes components for accessing a great variety of sources ('source adaptors'), a component to combine data from different sources ('data mashup' component), and components for building the graphical interface ('widgets'). The user builds graphically the mashup application by selecting, customizing, and interconnecting components. For example, a data mashup component is connected to the source adaptors of the sources to be combined.

To the best of our knowledge, all existing tools use the Pipes and Filters pattern [8] to express the data combination logic of the data sources making up the mashup application. This pattern follows a 'procedural' approach that forces the user to implement the combination logic in terms of a pipe. Depending on how the end-user needs to query the data, more than one pipe may need to be implemented to optimally execute the query. Unlike current tools, our data mashup component uses a 'declarative' approach based on the Data Federation pattern [22]. Combination logic is graphically expressed as relational operations (such as joins, unions, projections, selections, and aggregations) over the data sources to be combined. When the end-user launches a query, the data mashup component can automatically compute all the possible execution plans to solve it and let the optimizer choose one. Conceptually, each execution plan can be seen as a pipe in the Pipes and Filters pattern.

Our architecture also makes emphasis in component reusability. Source adaptors and data mashup components implement the same RESTful [5] interface. Apart from the advantages intrinsic to the RESTful architectural style, this allows, for example, reusing a data mashup component as a source component. It also makes possible the construction of 'template' widgets that can analyze the meta-information (a WADL specification [21]) of a component and provide specific functionality at run-time.

Finally, unlike other proposals, we reuse standard Java portal technology to implement the graphical interface. In particular, widgets are implemented as portlets [12][15]. At the graphical interface level, a mashup application is implemented as a portal page composed of a number of widgets. Some of these widgets will be connected to data mashup components. Reusing standard Java portal technology allows aggregating heterogeneous widgets in a mashup application and using the portal's event bus to coordinate them.

The rest of the paper is structured as follows. Section 2 presents a running example to illustrate our approach to building enterprise mashup applications. Section 3 provides an overview of the mashup tool architecture. Sections 4 and 5 discuss the design of the architecture. Section 6 discusses related work. Finally, Section 7 presents conclusions and outlines future work.

2 A Running Example

This section presents a running example[1] to exemplify our approach to building enterprise mashup applications. In the example, Martin, a Sales Manager from Acme, Inc. needs to perform a typical task: discovering new potential customers. Acme, like many other companies, uses an on-demand CRM to store data about its customers. In particular, Acme uses salesforce.com (http://www.salesforce.com). Customer's data includes, apart from general company information, the satisfaction level with Acme products and the name of a business contact in such a company.

Martin, like many other professionals, uses LinkedIn (http://www.linkedin.com) to maintain information about their business contacts. LinkedIn is a business-oriented social network that does not only allow maintaining information about personal contacts but also to access contacts from other users. LinkedIn users may specify public information, such as, personal information (name and title) and general information about her/his company (name, industry, and address), and private information (e.g. phone and email). Another LinkedIn user can see public information of the contacts of any of her/his direct contacts.

To discover new 'leads' (i.e. potential customers), Martin intends to use the following strategy:

- Search current customers in salesforce.com. He may be interested in retrieving these customers by different criteria, such as satisfaction level, industry, geographic location, or even by the name of a business contact. For this example, we will assume that Martin wishes to find customers by satisfaction level and/or business contact name. For each customer he needs its satisfaction level and the name of the business contact. These business contacts are among the Martin's direct contacts in LinkedIn. We will call 'reference contacts' to these business contacts.
- For each reference contact, search in LinkedIn her/his business contacts (the 'leads'). Many of them will work in other companies and Martin intends to contact them by using the reference contact. Martin may want to analyze all the leads or only those fulfilling a set of criteria. For this example, we will assume that he may want to restrict the returned leads to those working in a company of a given industry.
- To perform a global analysis, Martin wants to classify all lead companies by industry, using a pie chart summary.
- Finally, for each lead company, Martin wants to locate it in a map to plan a visit (e.g. by using Google Maps) and obtain information about it (e.g. by using Yahoo! Finance).

Since the number of customers in salesforce.com (and LinkedIn network) is large and variable, Martin quickly realizes this task is large and tedious. Clearly, he needs an application that fully automates the task. Martin is not a programmer. In consequence, it is not feasible for him to build such an application. Even for a programmer, building this application is not easy since it involves the access to five

[1] We have implemented a running prototype of the architecture proposed in this paper and we have used it to create the mashup illustrated in this section. The mashup can be accessed at http://www.tic.udc.es/mashup-demo.

applications with heterogeneous, complex interfaces. Martin could solve his need using our mashup tool. This tool provides a set of generic, reusable components that can be customized and assembled to engineer mashup applications.

The first aspect he needs to resolve consists in obtaining the data. For each lead, Martin wishes to obtain:

- Data about the lead (name and title) and her/his company (name, industry, and address). This information is provided by LinkedIn.
- Data about the reference customer: satisfaction level and reference contact. This information is provided by salesforce.com.

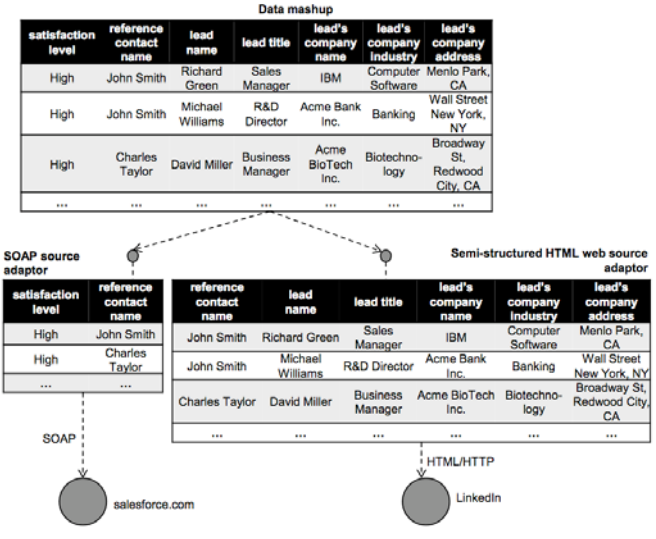

Fig. 1. Assembly of components to get the data of potential new customer companies

Fig. 1 shows how Martin solves the first aspect. The mashup tool provides 'source adaptor' components to access many kinds of sources. Source adaptors are non-visual components that allow accessing a source by providing a uniform interface to other components. Among others, the tool provides adaptors for data sources such as SOAP and semi-structured HTML web sources. All these data source adaptors allow seeing a data source as a web service that can be consulted through the uniform interface. The first one inspects a WSDL specification and invokes the appropriate operations. The second one uses web wrapper generation techniques to automate queries on a semi-structured HTML web source.

Martin selects the SOAP salesforce.com adaptor and configures it to obtain the data he needs. Then, he configures the semi-structured HTML web adaptor to automatically login with Martin's credentials in LinkedIn, fill in the search form with a reference contact name, navigate to the pages containing the public information of the leads and extract the target data.

To combine the data provided by salesforce.com and LinkedIn, Martin selects a 'data mashup' component and connects it to the previous two components. This component combines the data provided by both components, conceptually performing

a JOIN operation by reference contact name. The data mashup component is also a non-visual component that allows seeing the resulting data as a web service that can be consulted through the same interface as the data source components.

The second aspect Martin has to deal with consists in building the graphical interface of the application. The interface consists in: (1) a form to query leads according to Martin's criteria, (2) a pie chart for classifying leads by industry, (3) a map to locate lead companies, and (4) a web clipping of Yahoo! Finance to obtain the information about them. The mashup tool provides generic visual components that fulfill Martin's requirements. He only needs to customize and to assembly them. We name 'widgets' to these visual components. Fig. 2 shows the assembly of widgets making up the graphical interface.

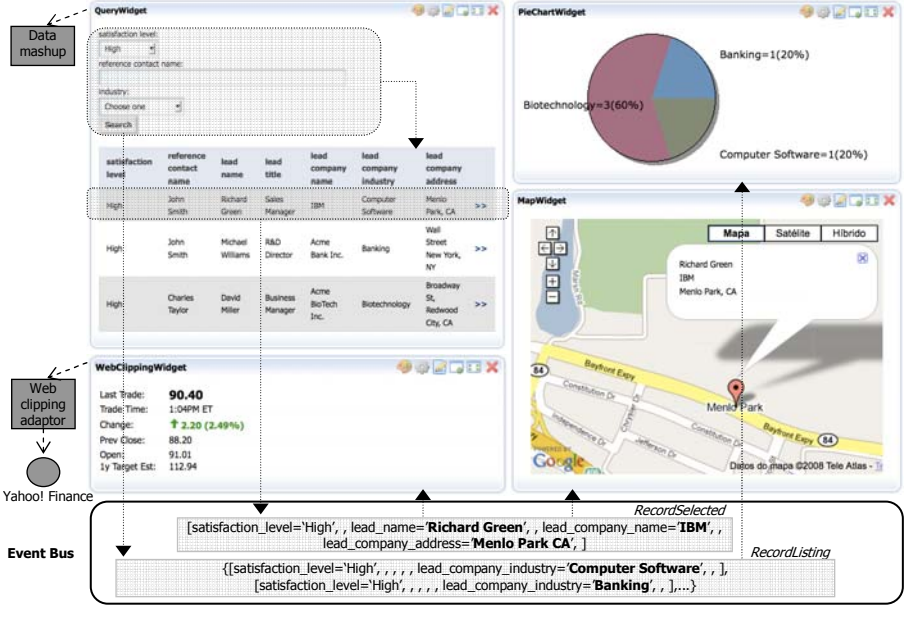

Fig. 2. Assembly of widgets in the example mashup

To provide the form for querying leads, Martin selects the 'query widget' and connects it to the data mashup component. This widget can analyze the meta-information provided by the data mashup component to find out its query capabilities (e.g. query parameters) and the schema of the returned data. From this information, it automatically generates an HTML form to allow querying the data mashup component, showing the results in an HTML table. In the example, the form has the following fields: satisfaction level, reference contact name, and industry.

To include the pie chart, Martin chooses the 'pie chart widget'. This widget generates a pie chart from a list of records using the values of one of the fields to classify them. In this case, Martin specifies this widget to receive the list of records displayed by the query widget and classify them by the industry field. How the pie chart widget receives the list of records? One approach would consist in Martin to

explicitly create a link between the query widget and the pie chart widget. However, in the general case this could result in the user would have to create manually a network of links between all widgets sending a given type of object and all the widgets wishing to receive it. To overcome this problem, widgets communicate each other by using an event-based model. An event has a logical name (e.g. *RecordListing*) and data (e.g. the list of records). A widget subscribes to an event by specifying its name. Each time a widget publishes an event, the mashup tool delivers it to all subscribing widgets. In this case, the query widget generates the *RecordListing* event when it obtains the results and the pie chart widget is prepared to receive events containing a list of records. Martin personalizes the pie chart widget to subscribe it to the *RecordListing* event. The query widget also allows selecting one of the records in the result listing. Selecting a record causes the query widget to send the *RecordSelected* event containing the selected record.

To locate lead companies on a map, Martin selects the 'map widget'. This widget receives an event containing a record that must include an address field. This widget uses Google Maps to show the information contained in the record into a map. Martin personalizes the widget to subscribe it to the *RecordSelected* event and to specify the name of the field containing the address. This way, whenever the user selects a record in the query widget, its information is automatically placed in the map.

Finally, to get the clipping of a given company from Yahoo! Finance, Martin chooses the 'web clipping widget'. This widget allows accessing a web page to extract a block of markup. Since accessing a web page involves automatic navigation (maybe traversing several forms and links), this widget must be connected to a 'web clipping source adaptor'. Like a 'semi-structured HTML web adaptor', this adaptor performs automatic navigation. However, unlike that adaptor, a web clipping adaptor does not extract data but a block of markup (without interpreting its meaning). In consequence, Martin selects a web clipping adaptor and configures it to access Yahoo! Finance. He also customizes the web clipping widget to use the web clipping adaptor. Like the map widget, the web clipping widget is also prepared to receive an event containing a record that must include a field for the company name. Martin subscribes the web clipping widget to the *RecordSelected* event and specifies the name of the field containing the company name.

3 Overview of the Mashup Tool Architecture

Fig. 3 shows the architecture of our mashup tool. The design of the architecture breaks down the functionality in 'layers' and 'common services'. Layers represent main functional blocks and have the traditional meaning in software design. Common services represent services that are useful for several layers.

The goal of the 'Source Access Layer' is to let the higher level layers access heterogeneous sources by using a common interface. As justified in section 4, we use a RESTful interface. Internally, this layer provides a 'source adaptor' component for each possible type of source. Each adaptor maps the meta-model and access modes used by the underlying source to the RESTful interface. Our current implementation provides adaptors for accessing typical data sources, such as SOAP/REST web services, relational databases, and semi-structured HTML web pages. It also includes

a special adaptor, ‘web clipping source adaptor’, to extract a block of markup of a HTML web source.

The ‘Data Mashup Layer’ allows defining ‘views’ combining one or several sources, which are accessed through the Source Access Layer interface. A ‘view’ has the same meaning as in databases. It is possible to define views as joins, unions, projections, selections, and aggregations of sources. As justified in section 4, we have chosen the Data Federation pattern to declaratively define the views. To this end, this layer provides the ‘data mashup’ component. The user connects this component to the source adaptors she/he needs. This layer (and in consequence the data mashup component) exposes the same RESTful interface to the higher level layer as the Source Access Layer. This allows reusing a data mashup component as a source adaptor, enabling to connect it to other data mashup components.

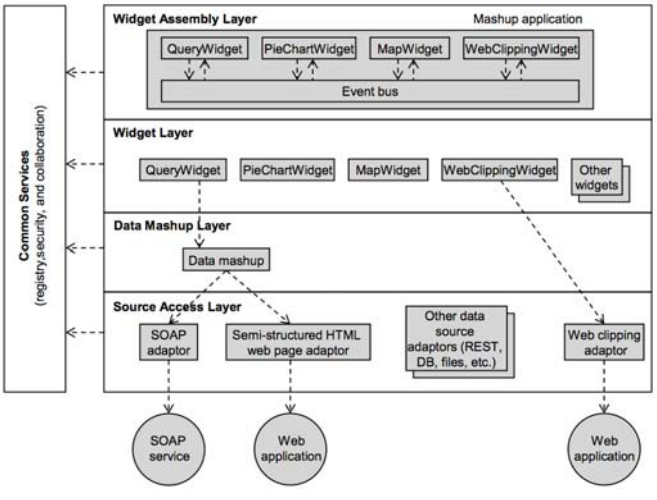

Fig. 3. Mashup tool’s architecture

To support the construction of the graphical interface of the mashup application, the ‘Widget Layer’ provides visual components, which we call ‘widgets’ (query widget, pie chart widget, etc.). Some widgets are connected to data mashup components (e.g. the query widget). As explained in section 5, we have decided to use standard Java portal technology to implement this layer and the layer above it. In particular, widgets are implemented as portlets [12][15].

The ‘Widget Assembly Layer’ allows assembling several widgets to create the final, unified graphical interface of the mashup. Assembly capability implies ‘aggregation’ and ‘coordination’ of widgets. Aggregation refers to the capability of adding heterogeneous widgets into a mashup application. Coordination refers to enabling communication between widgets inside the mashup application.

Finally, our architecture distinguishes three services, namely ‘registry’, ‘collaboration’, and ‘security’ to facilitate the implementation of all layers. Our current implementation includes support for registry and security services. Basic

registry capabilities include registering and searching components (individual sources, data mashups, widgets, and mashup applications). Additional registry capabilities include component lifecycle management (e.g. versioning) and community feedback (e.g. tagging, rating, etc.). The security service follows a role-based policy to specify and control the actions that can be taken on components. Finally, collaboration refers to the possibility of reusing knowledge automatically obtained by the tool in function of previous user interactions. This knowledge may be used to automatically or semi-automatically compose data mashups and widgets. We are currently working on adding this service to our tool.

4 Design of the Source Access and Data Mashups Layers

4.1 Access to Web Sources

In the Source Access Layer, semi-structured web sources require special attention. By semi-structured web sources, we refer to websites which do not provide any programmatic interface to access their contents and/or services.

In this case, this layer needs to use either web wrapper generation techniques [2][14] or web clipping techniques [1] to emulate a programmatic interface on top of the existing website. Web wrappers extract data from the target website in structured form. For instance, in our example, a web wrapper is used to automate the search for the contacts of a given reference customer in LinkedIn, extracting the structured data embedded in the HTML response pages. Our current implementation supports graphical generation of wrappers using the techniques proposed in [14].

In turn, web clipping automates the process of extracting target markup fragments from one or several pages in a website, providing reuse at the UI-interface level. In our example, the web clipping widget introduced in section 2 uses the web clipping adaptor provided by the Source Access Layer to obtain the desired Yahoo! Finance information. The web clipping adaptor automates the process of querying Yahoo! Finance (using the same automatic navigation techniques used for wrapper generation) and return the HTML markup corresponding to the fragment of the page containing the target information.

4.2 Interface Provided for the Source Access and Data Mashup Layers

Our approach reuses the RESTful architectural style [5] for the interface provided by the Source Access Layer and Data Mashups Layer. A RESTful web service allows accessing resources by using the standard HTTP methods GET, POST, PUT, and DELETE. Each resource is uniquely identified by a global URI. A RESTful interface presents several relevant advantages in the mashup environment:

- It supports both sources returning visual markup (such as HTML web sources used for web clipping) and sources returning structured data.
- Each individual data item obtained from a source/data mashup can be modeled as a resource accessible by a unique global URI. For instance, let us assume for the sake of simplicity that the fields *reference_contact_name* and *lead_name* uniquely identify an item returned by the data mashup of Fig. 1; then we could use a

URI such as *http://www.tic.udc.es/mashup_demo/leads?reference_contact_name=John+Smith&lead_contact_name=Richard+Green* to reference the lead contact named *Richard Green* obtained through the reference contact named *John Smith*.

- These resources can be read, updated, inserted, or removed by respectively issuing GET, POST, PUT, and DELETE HTTP requests on the URI of the resource. Of course, some resources may not admit some operations or may require authorization before executing them. Special URIs (e.g. *http://www.tic.udc.es/mashup_demo/leads?reference_contact_name=John+Smith*) can be constructed for executing queries by using the GET method.
- It promotes reuse at the resource granularity. Since each resource is identified by a global URI, any data mashup can link to a resource from another one just by including its URI. For example, let us suppose that, after creating the mashup application of section 2, Martin creates another mashup application to support tracking of the sales opportunities opened by using the first one. This second application could be based on a new data mashup gathering information from several sources and including in each opportunity a link to the original information that provoked it. For instance, if an opportunity came from the lead *Richard Green* of the reference contact *John Smith*, the new data mashup could include a link to *http://www.tic.udc.es/mashup_demo/leads?reference_contact_name=John+Smith&lead_contact_name=Richard+Green* in the returned information about the opportunity. Notice also how using links lets the higher layer or the application decide in an item-by item basis whether to pay the cost of obtaining the detail information or not.
- Using the unified interface provided by HTTP methods, lets third-party providers (proxies) transparently provide additional services (e.g. security, caching, etc.) to the clients invoking the sources and/or the data mashups.

To allow clients and higher layers to know the capabilities of a source/data mashup, we use the WADL language [21]. WADL[2] allows describing the resources offered by a RESTful web service, how they are connected, the data schema of the information associated to them, the HTTP methods supported by each resource and the parameters that can be used to invoke them.

4.3 Model Used for Expressing Data Combination Logic

The model used in the majority of mashups platforms to express data combination logic is based on the Pipes and Filters pattern [8]. A 'pipe' is composed of a series of interconnected components. Each component performs a specific operation such as accessing to sources, filtering input records, and/or merging input feeds. A connection from the component 'A' to the component 'B' denotes that the output from 'A' is the input to 'B'. Connections in such a model may be of several types and the most common are serial, joins, and forks. Components begin their execution when they have all their inputs available. This way, processing is propagated through the flow. The most commonly mentioned advantage of using this approach is simplicity in

[2] See http://www.tic.udc.es/mashup-demo/leads.wadl for the WADL specification of the data mashup shown in Fig. 1.

modeling these flows. It is assumed that pipes can be created by 'power users' which do not necessarily have programming skills.

However, we advocate for the Data Federation pattern to express data combination logic. This pattern has been extensively researched during the last decade to combine heterogeneous data. In this approach, the data combination logic is expressed as a 'view'. The view definition is written as a query (in SQL, for instance) over the data source schemas. Combination is expressed as relational operations like joins, unions, projections, selections, and aggregations. A graphical interface can allow users to create the view by interconnecting components that represent basic combination operations. This way, power users without programming skills can define views.

An important advantage of Data Federation pattern is its declarative nature. At run-time, when the user launches a query, the system automatically computes all the possible execution plans to solve it. The user can either manually choose the plan or let the system optimizer make the choice. The query capabilities supported by each source are also taken into account when generating the execution plans, so only the plans allowed by the source are generated. For instance, the LinkedIn source used in the example from section 2 requires a mandatory input parameter (the name of the person we want their contacts to be retrieved); therefore, only the execution plans satisfying that restriction will be generated for the data mashup in Fig. 1. To compute query plans having query capabilities into account and to choose the most optimal plan, we leverage on the techniques proposed in [17] and [7], respectively. Query capabilities of the sources are discovered at run-time using their WADL descriptions.

In turn, the Pipes and Filters pattern uses a 'procedural' approach: the system executes the pipe exactly as it was designed, following the flow path as defined. In fact, a pipe can be seen as the explicit definition of a particular query execution plan.

Let us go back to the running example introduced into the section 2 to highlight the differences between the Pipes and Filters pattern and the Data Federation pattern. Let us consider two different use case examples for Martin:

- **Example A:** Get all the leads that can be obtained from the *reference_contact_name John Smith*. In this case, the optimal way to solve it is by querying salesforce.com and LinkedIn in parallel using *John Smith* as the input parameter and merging the obtained records.
- **Example B:** Get all the leads that can be obtained from customers with high satisfaction level. In this other case, the system would query salesforce.com for contacts with a high satisfaction level. Then, for each customer retrieved, the system should search LinkedIn by using the *reference_contact_name* as the input parameter to obtain the data of the lead. The two sources need to be accessed sequentially because the query does not provide us with a value to fill in the mandatory input parameter of the LinkedIn data source.

Using the pipes and filter model, two different pipes would have to be created, each of which would solve one of the cases. The resulting pipes are shown in Fig. 4.

In turn, using the Data Federation pattern and given the source adaptors for salesforce.com and LinkedIn, the required data mashup can be modeled as a 'view' defined by a join operation between both data sources by the *reference_contact_name* attribute.

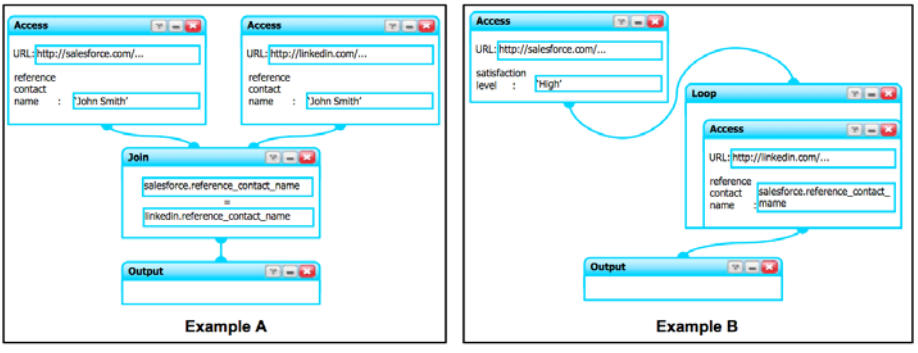

Fig. 4. Pipes and Filters approach

```
CREATE VIEW leads AS
  SELECT satisfaction_level, reference_contact_name,
    lead_name, lead_title, lead_company_name,
    lead_company_industry, lead_company_address
  FROM salesforce INNER JOIN linkedin ON
    salesforce.reference_contact_name =
    linkedin.reference_contact_name;
```

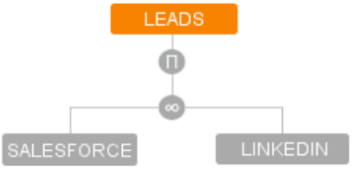

Fig. 5. Data Federation approach

Fig. 5 shows both the graphical representation of the data view created and its internal definition written in the SQL-like language used in our implementation to define views. Notice that in our implementation the user does not need to use SQL; the views are created using graphical wizards. The SQL expression defines an inner join between the views *linkedin* and *salesforce* (both exported by the Source Access Layer). Both example use cases shown above can be resolved by querying the 'Leads' view with queries such as:

```
select * from leads where reference_contact_name = 'John Smith'
select * from leads where satisfaction_level = 'High'
```

Notice that the user does not need to write these concrete queries at data mashup generation time. The data mashup component generates the WHERE clause automatically at run-time when it is accessed by the GET method, adding a condition for each parameter received in the request. Next, the data mashup component computes all possible execution plans and chooses the best one [7].

5 Design of the Widget and Widget Assembly Layers

Instead of creating yet another widget technology, we have reused standard Java portal technology [12][15] to implement individual widgets as portlets and mashup applications as portal pages. Portlets are interactive, web mini-applications which can

be aggregated into portal pages. Any standards-compliant portlet (widget), local or remote to the portal, can be aggregated into a page. Portlets communicate each other in a decoupled way through the portal's event bus.

The meta-information provided by the WADL specifications of the data mashup instances enables the construction of 'template' widgets. Unlike 'specific' widgets, such as the pie chart or map widgets introduced in section 2, template widgets do not implement specific business functionality. Instead, they acquire business value at run-time. One such widget is the query widget mentioned in section 2. This widget analyzes the WADL specification of the mashup component it is connected to. In view mode, it discovers the query capabilities of a data mashup instance and generates a form containing a field for each query parameter. The WADL specification also specifies the schema (e.g. an XML Schema, RelaxNG, etc.) of the returned data. The schema is used to display the results (a list of records) in an HTML table. To connect an instance of the query widget to a particular data mashup, the user selects the edit mode and specifies the URL of its WADL specification.

Fig. 6. MapWidget's edit mode

Another important design aspect to make possible the construction of reusable widgets consists in treating events in a generic way, so that can be consumed by any interested widget. On the one hand, data contained in events must be modeled in a generic way. Taking up the example of Section 2, the query widget generates a *RecordListing* event when obtains the results. This event contains an array of *Record*. Each *Record* is an array of *RecordField*, being a *RecordField* a data structure containing the name and the value of an individual field in a record. The *RecordListing* event can be consumed by any widget expecting to receive an event containing an array of records, such as the pie chart widget. In the same way, when the user selects a particular record in the query widget, an event containing the record is generated, which can be consumed by any widget expecting to receive a record, such as the map and web clipping widgets.

On the other hand, it is necessary to provide a mechanism to subscribe a widget to an event and specify how to process it. To subscribe a widget to an event, it is necessary to specify the name of the event by using the portal capabilities. The widget can provide a wizard in the edit mode to specify how to process the event. For example, the map widget in the edit mode (see Fig. 6) lets the user specify the name of the field (in the record contained in the event) containing the address (needed to obtain the map coordinates) and the names of the fields to display in the 'balloon'. This way, the consumer widget (the map widget, in this case) does not assume particular names for the fields of the events it receives, improving its reusability.

6 Related Work

In the commercial arena, tools like Yahoo! Pipes [23] and Microsoft PopFly [13] are oriented to the construction of consumer mashups. They offer hosted tools that provide a set of user-configurable modules to create pipes to process RSS/Atom feeds and other types of REST web services. They also provide some support to show the output of the pipe using pre-created template widgets such as maps or photo albums. Therefore, the functionality they provide fits into the Data Mashup Layer and Widget Layer of our architecture; they do not provide support neither for assembling several widgets in a more complex mashup nor to access other kinds of data sources (such as databases or semi-structured web sources). Other key differences with our proposal include that they use the Pipes and Filters pattern to specify the data combination logic and they do not provide a standard interface between layers. Besides, since these tools are oriented to consumer mashups, the processing and displaying components they offer are not designed for enterprise environments.

There also exist a number of commercial tools that are exclusively oriented to enterprise data mashups. For instance, RSSBus [19] provides simple ESB (Enterprise Service Bus) functionality based on the Pipes and Filters pattern to generate mashups from a variety of enterprise sources such as REST web services, databases, content servers or CRM tools. All those source formats are imported as RSS feeds, which is the format that the platform natively uses. With respect to our proposal, on one hand, they do not deal with user interface issues; on the other hand, they rely on the Pipes and Filters pattern instead of the Data Federation pattern.

Like our proposal, JackBe [10] and IBM Mashup Center [9] provide functionality for creating both ‘data mashups’ and ‘graphical user interfaces’. The architecture of both tools is similar: they use the Pipes and Filters pattern for data combination, support creating widgets from data combination components (similarly to our ‘template’ widgets), and allow assembling several widgets to build the graphical interface of the mashup application. All the languages and interfaces used by these tools are proprietary. Unlike these tools, we (1) opt for the Data Federation pattern instead of Pipes and Filters pattern, (2) rely on RESTful interfaces between layers, and (3) reuse standard portal technology.

Like our proposal, the Yahoo! Query Language (YQL) platform [24] allows querying web sources by using an SQL-like language. However, this platform only allows getting data from one individual web source. It is possible to query additional web sources, but only in nested queries to filter the values of the main web source by using the IN operator (like in SQL). In consequence, this platform could not be used to implement the data mashup component of our architecture. Furthermore, its approach is not truly declarative, since each YQL query has only one possible execution plan.

In academia, [25][3] present a framework to assembly several independently created widgets. The underlying model used is also event-based: widgets emit events in response to user actions; the events have attached meta-information that can be used to fill in the input parameters of the operations exposed by other widgets subscribed to the event. [6] and [11] also present proposals for event-based widget inter-communication. The first one is based on the OpenAJAX initiative [16], while the second one uses proprietary mechanisms. Neither of these proposals addresses the

problem of data combination (the Data Mashup Layer of our architecture). Their proposals for widget inter-communication are conceptually similar to ours.

[20] proposes an approach for building mashups by example. Firstly, the user provides examples for the system to be able to create wrappers and perform transformations on the extracted data. Secondly, the system suggests the user to join different wrappers by detecting overlaps in the values extracted by them. The simplicity of use comes at the cost of powerfulness: the range of possible transformations is limited and the only way available to combine the data is using simple equijoin operations. In addition, the system only allows showing the results in an HTML table, and does not provide any way to combine several widgets.

MARIO [18] combines 'tagging' and automatic composition techniques to allow creating automatically pipes from a set of keywords ('tags') specified by the user. Although using automatic composition techniques is a promising idea, MARIO is limited to use RSS feeds as data sources and uses the Pipes and Filters pattern for data combination. Furthermore, it does not address the graphical interface.

Mash Maker [5] is another system that allows creating mashups without needing to have any technical knowledge. The system assumes that the user will create extractors for web sources and widgets to manipulate and visualize data. As normal users browse the web, Mash Maker suggests widgets that can be applied on the current page based on the experience from previous users. With respect to our proposal, Mash Maker uses a client-side approach heavily based on collaboration that fits better with consumer mashups than with enterprise mashups. In addition, the available components to extract, transform, and manipulate data are quite limited.

7 Conclusions and Future Work

In this paper we have presented the design of an enterprise-oriented, component-based mashup tool that fully supports the construction of mashup applications. The user builds graphically the mashup application by selecting, customizing, and interconnecting components. With respect to other tools, our approach presents the following advantages: (1) we use a declarative approach to express data combination logic based on the Data Federation pattern; (2) we use a RESTful approach to define the interface of source adaptors and data mashup components, which improves system flexibility; and (3) we reuse standard portal technology to improve reusability and interoperability at the graphical interface level.

We are now exploring automatic composition of data sources and the application of collaborative filtering techniques to suggest additional data sources and widgets during the mashup creation process.

References

1. Bellas, F., Paz, I., Pan, A., Diaz, O.: New Approaches to Portletization of Web Applications. In: Handbook of Research on Web Information Systems Quality, 270–285 (2008) ISBN: 978-1-59904-847-5
2. Chang, C.-H., Kayed, M., Girgis, M.R., Shaalan, K.F.: A Survey of Web Information Extraction Systems. IEEE Transactions on Knowledge and Data Engineering 18(10), 1411–1428 (2006)

3. Daniel, F., Matera, M.: Mashing Up Context-Aware Web Applications: A Component-Based Development Approach. In: Proceedings of the 9th International Conference of Web Information Systems Engineering, pp. 250–263 (2008)
4. Ennals, R.J., Brewer, E.A., Garofalakis, M.N., Shadle, M., Gandhi, P.: Intel Mash Maker: Join the Web. SIGMOD Record 36(4), 27–33 (2007)
5. Fielding, R.T., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology 2(2), 115–150 (2002)
6. Gurram, R., Mo, B., Gueldemeister, R.: A Web Based Mashup Platform for Enterprise 2.0. In: Proceedings of the 1st International Workshop on Mashups, Enterprise Mashups and LightWeight Composition on the Web (MEM & LCW), pp. 144–151 (2008)
7. Hidalgo, J., Pan, A., Alvarez, M., Guerrero, J.: Efficiently Updating Cost Repository Values for Query Optimization on Web Data Sources in a Mediator/Wrapper Environment. In: Etzion, O., Kuflik, T., Motro, A. (eds.) NGITS 2006. LNCS, vol. 4032, pp. 1–12. Springer, Heidelberg (2006)
8. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, Reading (2003) ISBN: 032120068
9. IBM Mashup Center, http://www.ibm.com/software/info/mashup-center
10. JackBe, http://www.jackbe.com
11. Janiesch, C., Fleischmann, K., Dreiling, A.: Extending Services Delivery with LightWeight Composition. In: Proceedings of the 1st International Workshop on Mashups, Enterprise Mashups and LightWeight Composition on the Web (MEM & LCW), pp. 162–171 (2008)
12. Java Community Process: Java Portlet Specification - Version 2.0, http://jcp.org/en/jsr/detail?id=286
13. Microsoft Popfly, http://www.popfly.com
14. Montoto, P., Pan, A., Raposo, J., Losada, J., Bellas, F., Carneiro, V.: A Workflow Language for Web Automation. Journal of Universal Computer Science 14(11), 1838–1856 (2008)
15. OASIS: Web Services for Remote Portlets Specification – Version 2.0, http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec-os-01.html
16. OpenAjax, http://www.openajax.org
17. Pan, A., Alvarez, M., Raposo, J., Montoto, P., Molano, A., Viña, A.: A Model for Advanced Query Capability Description in Mediator Systems. In: Proceedings of 4th International Conference on Enterprise Information Systems, ICEIS, vol. I, pp. 140–147 (2002)
18. Riabov, A.V., Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A.: Wishful Search: Interactive Composition of Data Mashups. In: Proceedings of the 17th International Conference on World Wide Web, pp. 775–784 (2008)
19. RSS Bus, http://www.rssbus.com
20. Tuchinda, R., Szekely, P., Knoblock, C.: Building Mashups by Example. In: Proceedings of the 13th international conference on Intelligent User Interfaces, pp. 139–148 (2008)
21. Web Application Description Language, https://wadl.dev.java.net
22. Wiederhold, G.: Mediators in the Architecture of Future Information Systems. IEEE Computer 25(3), 38–49 (1992)
23. Yahoo! Pipes, http://pipes.yahoo.com/pipes
24. Yahoo! Query Language, http://developer.yahoo.com/yql
25. Yu, J., Benatallah, B., SaintPaul, R., Casati, F., Daniel, F., Matera, M.: A Framework for Rapid Integration of Presentation Components. In: Proceedings of the 16th World Wide Web Conference, pp. 923–932 (2007)