Let It Flow: Building Mashups with Data Processing Pipelines

Biörn Biörnstad[1] and Cesare Pautasso[2]

[1] ETH Zurich, Computer Science Department, 8092 Zürich, Switzerland
[2] University of Lugano, Faculty of Informatics, 6904 Lugano, Switzerland

Abstract. Mashups are a new kind of interactive Web application, built out of the composition of two or more existing Web service APIs and data sources. Whereas "pure" mashups are built relying entirely on the Web browser as a platform for integration at the presentation layer, this approach is not always feasible and part of the mashup integration logic must be deployed on the Web server instead. In the case study presented in this paper, we explore a novel approach to build mashups out of heterogeneous sources of streaming data. In particular, we introduce a layered mashup architecture, with a clear separation between the mashup user interface and the mashup integration logic run by a mashup engine. To do so, we show how to build a mashup application that displays in real time the location of visitors connecting to a Website. The integration logic feeding a map widget by processing the Web site logs is developed using a data flow model that connects a set of reusable and heterogeneous components into a data processing pipeline. We conduct a brief performance evaluation of the mashup showing that the pipeline introduces minimal delay and discuss several extensions of the mashup.

1 Introduction

Software reuse and composition has reached the next evolutionary level with mashups, where software components delivered as a service on the Web and databases published as Web data feeds are combined in novel and unforeseen ways. Whereas the term mashup originates from the practice of sampling existing music and mixing it together, software mashups are typically associated with software integration performed at the User-Interface layer [1]. In modern Web applications, this translates to (mis)using the Web browser as an integration platform for running interactive Web applications made out of different existing Web sites, applications, services, and data sources [2]. Given the limitations of such "pure" *client-based* mashups, which make it difficult if not impossible for the browser to safely interact with a variety of distinct service providers, complex mashups are designed using a *server-based* architecture. This alternative design separates the integration logic from the presentation of the mashup to better fit the constraints of current Web application architectures.

In this paper we present a novel approach, based on data processing pipelines, for the development and execution of the integration logic of server-based

E. Di Nitto and M. Ripeanu (Eds.): ICSOC 2007 Workshops, LNCS 4907, pp. 15–28, 2009.

mashups. Whereas the "pipe and filters" architectural style [3] goes back to early UNIX command line pipelines, it has seen a renaissance in recent mashup tooling (e.g., Yahoo Pipes [4], Microsoft Popfly [5], IBM DAMIA). Our mashup composition environment also promotes the notion of flow as the main construct to visually represent service and data integration pipelines. In addition, it introduces a generalized approach to composition that makes it applicable to mash up *heterogeneous* software and data services. We illustrate this with a case study showing how we built a concrete mashup application: a monitoring tool for plotting the location of Web site visitors on a map widget in real time. This mashup is built as a pipeline linking the access logs of a Web site with a geocoding service for IP addresses. The results are streamed to be displayed on a map widget running on the Web browser.

This paper is organized as follows. In Section 2 we define the challenges implied by the mashup application example. In Section 3 we present how to address these challenges with the integration logic for the monitoring application which is built by linking the mashup components into a pipeline. In Section 4 we discuss the architecture of the mashup runtime engine. In Section 5 we evaluate the performance of our approach regarding the ability to satisfy the real time requirement. In Section 6 we discuss the impact of several ideas for extending the mashup application. Section 7 discusses related work and Section 8 concludes the paper.

2 Mashup Example Challenge

Our case study focuses on a simple mashup application, used to track visitors of a Web site on a map (Figure 1). The mashup extracts the information from the access log of a Web server and integrates it with a geocoding service that attempts to provide map coordinates for each IP address found in the log. As opposed to similar mashups (e.g., GVisit [6]), which provide a daily update of the visitors map, we are interested in providing a real-time monitoring view.

In more detail, the mashup user-interface contains the map widget that displays the Web site visitors using markers located at the visitor's geographical coordinates. Each marker is also associated with the original IP address, so that

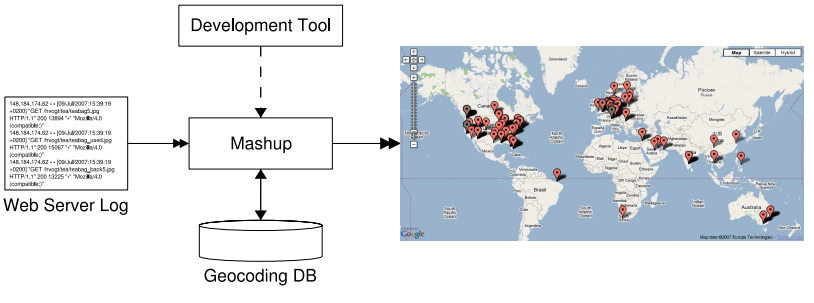

Fig. 1. Overview of the mashup example application

the user may find out more information about the Web log entry by clicking on the corresponding map marker. Markers appear on the map as soon as the Web site has logged the corresponding "hit" and the geocoding service has successfully estimated the geographical location of the visitor. Old markers are automatically removed. The user can configure how many markers should be kept on the map at the same time.

This case study highlights some of the most common integration challenges involved in building a mashup. These can be summarized as follows.

Data Extraction. Information relevant for the mashup is polled or pushed from a set of data sources. In our case, the Web site log must be monitored for changes and each new entry must be parsed to extract the IP address of the visitor.

Data Heterogeneity. Different data sources may present data in different formats, which require more or less effort to "scrape out" [7] the required information for providing it to the mashup. In our example, different Web servers may use different formats to store their logs, even though these are typically stored in a plain-text format (i.e., not in XML). Thus, each entry is separated by a line feed character, and the IP address of the visitor is contained at some predefined position within a log entry.

Data Integration. By definition, in a mashup, data from more than one source must be combined in some way. The example requires joining the data extracted from the logs with geocoding information stored in a database. A more complex form of integration would, for instance, attempt to classify visitors based on their activities while on the website (e.g., to distinguish successful file downloads from incorrect user authentication attempts).

Service Heterogeneity. Access to the data source may take place under different styles: a subscription to an RSS feed, a traditional Web service invocation using SOAP, the retrieval of the resource state of a RESTful Web service [8], or by piping the result of a UNIX command line. Regarding the example mashup, the geocoding database can usually be accessed remotely through a Web service interface [9,10,11] which may, however, put a daily cap to limit the number of messages exchanged. As a more efficient alternative, a JDBC interface may be available to directly query a locally installed copy of the database [12,13,14].

Data Quality. Due to limited coverage of the geocoding database, not all IP addresses can be successfully located and therefore it may be impossible to plot markers with the accurate position of all log entries. In general, the quality of the data from different sources may vary and not all service providers may be trusted to provide correct results.

Real-time Update. In a monitoring application, new data must be displayed on the mashup user-interface widgets as soon as it is generated. Thus, a streaming communication channel must be opened linking the various components of the mashup so that data can flow from the Web site visitor log on the Web server to the map widgets on the Web browser.

Maintainability. The long-term maintainability of a mashup is directly correlated with the amount of change affecting the APIs of the composed Web

services. No-longer maintained mashups are very likely to break as the underlying APIs evolve independently, and the data sources integrated by the mashups change their representation format. Having a clear model of the integration logic of the mashup helps to control the effect of changes. For example, when running the mashup example with a different Web server, only the data extraction part of the pipeline needs to be updated in case the access log format has changed.

Security. From a security standpoint, the mashup needs to be configured to gain access to the Web server logs. This requires entrusting the mashup with credentials that allow it to access the Web server via a secure shell connection. In general, it remains a challenge for users to safely delegate a mashup to access remote data sources and services on their behalf.

3 Mashup Integration Logic

The mashup application is designed using a layered architecture, separating the presentation from the integration logic. The former is implemented using standard AJAX techniques [15], used to create a map widget and to asynchronously fetch the data stream to be displayed on it. In this section we focus on how such a data stream is computed with our flow-based approach.

The integration logic can be developed bottom-up or top-down. When working *bottom-up*, the mashup components (i.e., Web services, data sources and glue code) with their interfaces and access mechanisms are defined before they are wired together in a pipeline. In a *top-down* approach, first the overall structure of the mashup is specified and then the details of each mashup component are worked out. Often, a mixed *iterative* approach is needed, because some of the components have a predefined interface that cannot be changed. Thus, *glue components* have to be introduced in order to make the existing components work together. In the following, we take a top-down approach to describe the mashup integration logic.

Our example mashup combines the log file of a Web server with a geolocation service. These have been integrated using the visual data flow pipeline shown in Figure 2. The model uses the following notation. The data flow is a graph consisting of *components* (rectangular boxes) which have *input and output parameters* (rounded boxes). Parameters are attached to their respective component with small hollow arrows. In abstract terms, a component represents a transformation on the input data received in the input parameters whose result is stored in the output parameters. A component can be implemented using a variety of mechanisms that allow it to communicate with Web services and data sources.

The flow of data in the pipeline is specified by drawing filled arrows between output and input parameters of components [16].

The data flow pipeline consists of three main components: readLog which accesses the Web server log, lookupCoordinates which looks up the geographical coordinates for a given IP address, and streamToBrowser which sends the results to the browser for visualization. In order to correctly integrate these

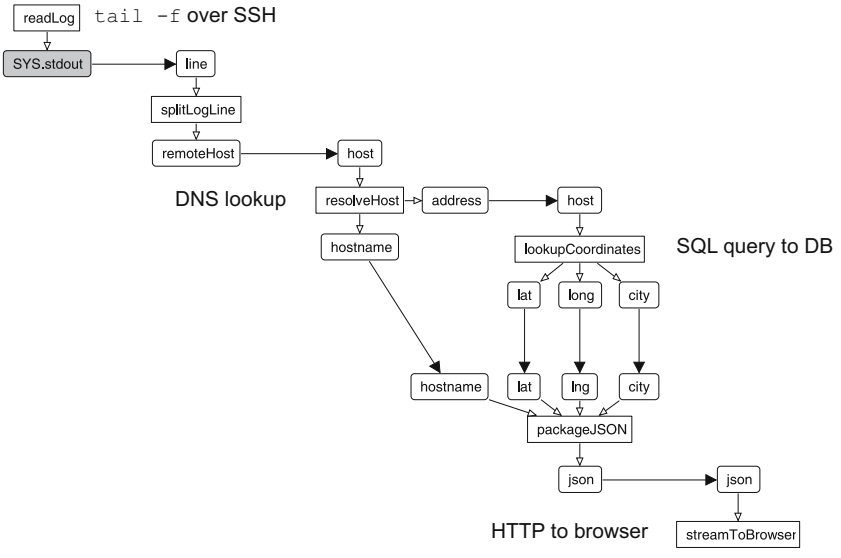

Fig. 2. The integration logic data flow pipeline for the mashup example

three components other glue components have been included to deal with data heterogeneity and quality issues. The implementation of all the components is summarized in Table 1 and is described in more detail in the rest of this section.

The readLog component retrieves the log from the Web server, one entry after the other. Since a Web server does not make its log file publicly available, the log file is accessed through an SSH connection to the server. There, to extract the data, the `tail -f` UNIX command is run to fetch new log entries. With it, the readLog component outputs the entries to be processed by the rest of the mashup as soon as they are appended by the Web server. If the information for the mashup were provided differently, readLog could use a different data extraction mechanism to access the data stream (e.g., RSS, XMPP [17]).

The log entries from the Web server consist of several fields separated by a space character as show in the following example.

```
148.184.174.62 - - [09/Jul/2007:15:39:19 +0200]
"GET /index.html HTTP/1.1" 200 13894 "-" "Mozilla/4.0 (compatible;)"
```

The most interesting field for this mashup is the first one. It contains the `remoteHost` – the address of the client visiting the Web site. In order to extract the relevant information, the splitLogLine component parses the log entry and provides the values of individual fields of the entry in its output parameters. splitLogLine encapsulates the knowledge about the log format and, should the log entry format change, it would be the only affected component. As shown in Table 1, the component is implemented with a regular expression, which allows for a fast parsing of the log entry.

Table 1. Component implementation details

readLog (SSH)
`tail -f logs/access_log`
splitLogLine (Regular expression)
`(\S+)\s+(\S+)\s+(\S+)\s+\[(.+)\]\s+"(.*)"\s+(\S+)\s(\S+)\s+"(.*)" ...`
resolveHost (Java snippet)
`...address = inetAddress.getHostAddress(); ...`
lookupCoordinates (JDBC)
`select lat, long, city from coords where subnet = network('%host%/24')`
packageJSON (String concatenation expression)
`{"lat": %lat%, "lng": %lng%, "city": %city%, "hostname": %hostname%}`
streamToBrowser (HTTP data port)
`<script>acceptData(%json%)</script>`

The next component of the mashup deals with the data heterogeneity of the information extracted from the log entry. Usually, Web servers store raw IP addresses in their logs to save the overhead of a reverse DNS lookup for every access to the server. However, a server might be configured with address resolution. To decouple the mashup from the actual server configuration, we introduce an additional component. The resolveHost component takes as input either a numerical IP address or a symbolic host name and ensures that its output always contains a raw IP address that fits with what is needed for looking up the corresponding geographical coordinates. Additionally, since we want to be able to display the host name of the visitor in the user interface, this component also does a reverse lookup to retrieve the DNS host name for the IP address (which, of course, is not necessary if this has already been done by the Web server). The host name resolution is implemented with a Java snippet, relying on the services of the DNS [18].

The conversion of the IP address to geographical coordinates is performed by the lookupCoordinates component. In the example mashup, we use hostip.info [9] as the data provider. The service can be used through its RESTful interface which returns coordinates in plain text or XML. However, for high-volume applications, the entire database can be downloaded and deployed locally. To optimize the performance of the mashup, lookupCoordinates accesses a local copy of the database through JDBC. The lookup is implemented as an SQL query to retrieve the name of the city and its coordinates for every IP address provided as input. Since the database contains an entry for every C-class subnet, the IP address (`%host%`) is converted before it is used in the `WHERE` clause of the query. Since the hostip.info database is not complete, IP addresses cannot always be translated to geographical coordinates. In such a case, no information can be sent to be displayed in the visualization part of the mashup running in the browser. This data quality problem is handled by skipping the

remaining components of the pipeline if no geographical information could be retrieved.

Before the coordinates can be sent to the browser, they need to be packaged in a suitable format. This is done in the packageJSON component. The format depends on the mechanism which is chosen to communicate with the JavaScript application running in the browser. In our case we chose to serialize the data using the JSON format [19]. To do so, we use a string concatenation expression, which replaces the names of the component input parameters (bracketed with %) with their actual value. The transfer of data is done by the streamToBrowser component. To ensure that the data triggers an update of the map widget we wrap the data into `<script>` elements which are executed by the browser as they arrive. The actual transfer of the data using HTTP is described in Section 4.

4 Layered Mashup Architecture

As shown in Figure 3, the mashup is designed with a layered architecture. Thus, it cleanly separates the user-interface part of the mashup, running in a browser as a rich AJAX application, from the integration logic, deployed in the mashup engine, and the Web services and data source providers, which are connected to the engine using different kinds of data ports. The mashup integration logic is developed using a visual environment presented at the end of this section.

4.1 Mashup Engine

The *mashup engine* is at the core of the architecture. This is were the different data sources are integrated and coordinated and new data products are made available to clients (like the browser or another mashup engine). As new data arrives, the engine is capable of streaming it through a mashup. To increase the throughput of the mashup, this is done in a pipelined fashion, i.e., multiple stream elements are processed in parallel, each in a different component of the

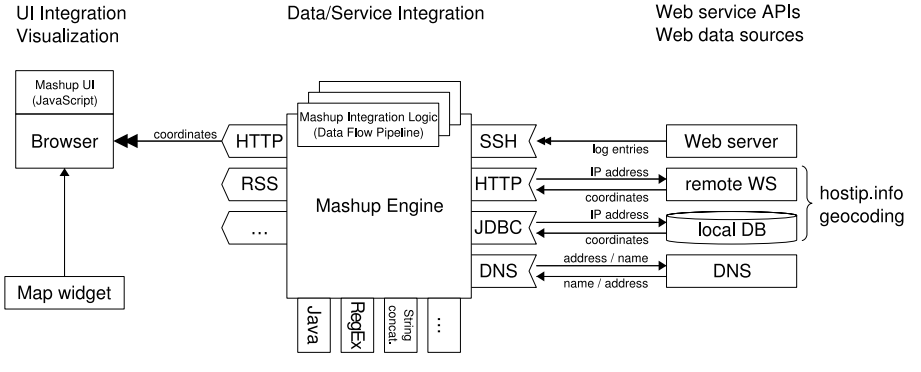

Fig. 3. Layered Mashup Architecture

mashup [16]. Different mashup data flow pipelines can be deployed to the engine by developers and multiple instances of the same mashup can run simultaneously (e.g., to monitor different Web server logs).

To control the execution of a mashup pipeline, the engine provides its own API to the mashup user interface. In the example, the user interface starts running a mashup pipeline and subscribes to its result stream. In a simpler case, the mashup integration logic can be invoked within a single request-response interaction.

In addition to its remote interaction capabilities, the engine also features the ability to invoke local components implemented using Java snippets, regular expressions, string concatenation expression, and XML manipulation techniques (such as XPath, or XSLT). In the example mashup, this is used to implement the glue components, for which the overhead of a remote interaction is spared.

4.2 Data Ports

The mashup engine interacts with external systems (Web services, data sources and clients) through *data ports*. Data ports provide a communication abstraction to deal with the service heterogeneity problem. At runtime, the components of a mashup pipeline are bound to a data port capable of translating service-specific protocols (e.g., HTTP, SOAP, RSS, SSH, JDBC) into the generic component model used in the mashup. Data ports are also used to share the results of the mashup pipeline with its clients, including the mashup user interface. Data ports can be classified by the interaction style they support. We use the following categories: *request-response*, *one-way*, and *streaming*, and further distinguish the direction of the communication (inbound or outbound) relative to the mashup engine. A flexible plug-in mechanism for data ports allows the mashup engine to easily support new protocols that fall within these interaction styles.

Request-response data ports. *Request-response* is probably the most commonly used interaction style on the Web, since HTTP is based on it. It also fits with many programming paradigms as it models the way synchronous operations, procedures, functions, or method calls behave. In the mashup engine, this style is used by data ports for service invocation (e.g., SOAP/HTTP, REST), but also to query a database (SQL/JDBC). When a component is implemented with an outbound request-response data port, its input parameters are used to construct the request to the external service. The response of the invocation is provided into the output parameters. The lookupCoordinates component of our example accesses the geocoding database through the JDBC data port which is outbound request-response style. The IP address received in the host input parameter is inserted into an SQL query that the data port sends to the database. From the response dataset, the data port extracts the city, lat and long columns into the corresponding output parameters of the component.

While outbound request-response is used to invoke services, inbound request-response can be used to provide the mashup as a service to other clients. To

do so, we follow an approach similar to WS-BPEL, where the mashup pipeline begins with a *receive* component and it finishes with a *reply* component.

One-way data ports. In contrast to request-response, in the *one-way* interaction style information is only transferred in one direction. This is the style used by messaging systems, and is also supported by SOAP when transported using message-oriented middleware. A component receiving a one-way message stores the message content into its output parameter. Vice versa, the input parameters of a component can be used to build the message to be sent out.

In the example, the results of the pipeline could be posted into a message queue with an outbound one-way message. However, this would require the mashup user interface on the browser to consume such messages from the queue. In order to simplify the communication between the engine and the mashup user interface, we have chosen a different solution based on a streaming HTTP data port, as discussed next.

Streaming data ports. Whereas the one-way interaction style focuses on the exchange of one and only one message at a time, streaming involves sending or receiving a potentially unbounded number of messages that are meant to be processed before the entire stream of messages is completed. In the context of our example mashup, *streaming data ports* are of particular interest since they make it possible to process data as soon as it is made available by its data source.

Streaming interactions can be accomplished in different ways. Ideally, data is *pushed* from the data source to the mashup engine over an open connection (e.g., XMPP, SSH). If this is not possible, the data port can use periodic polling to *pull* the data from the source as is done in the case of RSS. In both cases, a stream of messages can be identified by a certain context, e.g., a TCP connection or the URI of an RSS feed.

A stream of messages typically enters a mashup through the first component of the data processing pipeline (`readLog` in our example). The component provides the data in its output parameters from where it is forwarded to the rest of the pipeline. This model works for both push and pull interactions, because the data port abstracts from the low-level communication.

The data for our example mashup is retrieved over a streaming SSH data port. This data port opens an SSH connection to the remote system and submits a command for execution (e.g., `tail -f`). The response received from the remote program is split into individual lines as the output is streamed to the mashup engine. These lines are then forwarded to the corresponding component in the mashup as soon as they have been received. This allows the mashup to process one line at a time.

Usually, the last component of the mashup pipeline makes use of an outbound streaming data port. Data in the input parameters of the component is streamed out through the data port, e.g., using an established HTTP connection, or appended to an RSS feed which is polled by clients.

In our example, in order to transfer the marker coordinates to the browser, we use a streaming outbound HTTP data port. Since it is not possible for the

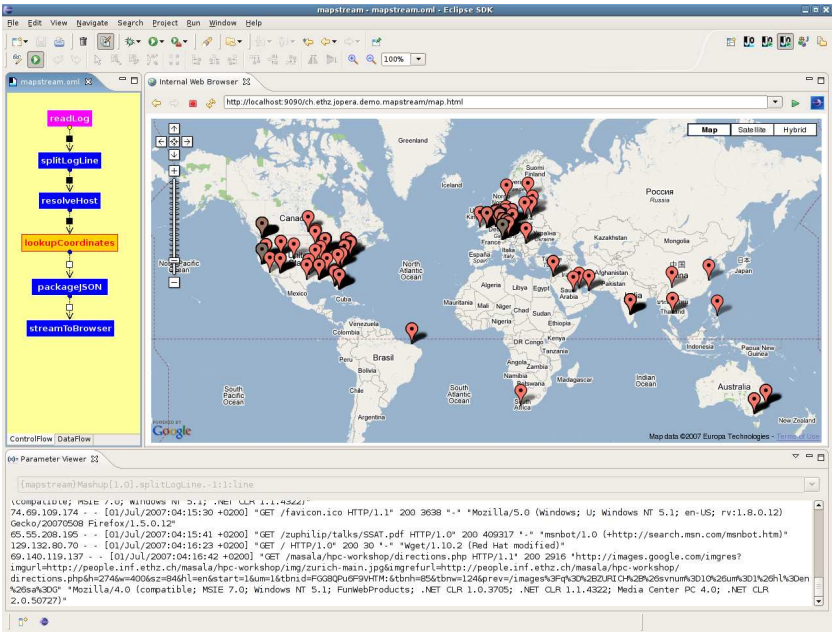

Fig. 4. Screenshot of the mashup example application running within the mashup development environment

mashup engine to open a connection back to the browser, it is the responsibility of the mashup user interface to initiate the connection to the HTTP daemon embedded into the data port. The URI of the HTTP request is used to specify from which mashup instance the stream of JSON coordinates should be received. The HTTP connection is kept open so the coordinates calculated by the mashup instance can be streamed – as part of an infinite HTTP response – directly to the corresponding user interface running in the browser.

4.3 Mashup Development Environment

The mashup engine is accompanied by the Eclipse-based JOpera [20] visual development environment to support the development cycle of the mashup integration logic. The environment supports the visual design and composition of the mashup data flow pipeline (Figure 2). Before a mashup pipeline is deployed in production, it can be debugged with the same graphical environment (Fig. 4).

Such an IDE for the mashup integration logic complements existing tools for the development of mashup user interfaces (e.g., [21]). For example, when the Eclipse Web Tools Platform (WTP [22]) is used in conjunction with JOpera, it becomes possible to provide an integrated mashup development environment for the visual design of the integration logic as a set of data flow pipelines as well as the development of the corresponding AJAX frontend to display the results.

5 Performance Evaluation

As the goal of our mashup application is to provide the user with a real-time update of the interface, we have evaluated the performance of the mashup pipeline. The goal of this experiment was to estimate whether a layered architecture is capable of delivering a real-time user experience for our monitoring application.

During the experiment the mashup engine, the geocoding database, and the browser were each running on a machine equipped with an Intel Pentium 4 processor with Hyper-Threading enabled running Linux 2.6 at 3GHz and with 1 GiB of main memory. The mashup engine was executed by Sun's Java HotSpot Server VM 1.4.2 and the geocoding database was served by PostgreSQL 8.1.9. The user interface was displayed by Firefox 1.5.0.7.

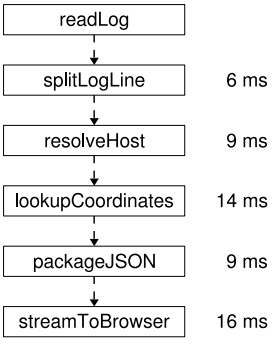

Fig. 5. Delay introduced by each pipeline component

In the experiment, we instrumented the pipeline to measure the processing time of each component (Figure 5). The readLog component has not been measured, because of the varying delay introduced by the buffering of the Web server when writing the log file. The processing inside the pipeline (from splitLogLine to packageJSON, inclusive) takes 41 ms. The delay for sending the resulting coordinates to the browser is 16 ms (including network latency for a local area connection). Thus, the total delay before a new Web request is reflected in the mashup user interface is less than 60 ms. Hence, our layered approach to mashup design is capable of fulfilling the real-time requirement of the monitoring application example.

6 Extending the Mashup Application Example

Even though the example application presented in this paper already provides useful functionality, in this section, we enumerate several ideas for extensions and discuss their impact on the mashup pipeline.

To enhance the markers on the map widget, we can display additional information: the time of the visit, the URL which was requested or the type of browser (*user agent*) which was used, by forwarding the appropriate data from the splitLogLine component to the packageJSON component.

Without having to alter the structure of the mashup pipeline, the monitoring application can be turned into a log analysis tool by simply replacing the main data source of the mashup. By changing the command executed by the readLog component from "`tail -f`" to "`cat`", the entire log file is streamed through the data flow pipeline and, thus, past requests are also processed and progressively visualized by the mashup.

As the geocoding database we use is based on a community effort, the contained information is not complete, i.e., it does not cover the location of all possible IP addresses. To increase the likelihood of a successful lookup, the mashup could be extended to query a commercial provider of geocoding information (e.g., [10,11]) if the address could not be found in the local database. To do this, a new component with the same input and output parameters is added in parallel to the existing lookupCoordinates component. The new component is then only triggered if lookupCoordinates does not return a result.

A more advanced extension is used to publish a subset of the annotated log entries in an RSS stream. Clearly, there is no point in providing the complete log over RSS. With the appropriate filtering component it is possible to provide notifications of "interesting" accesses to the Web server. This could include the detection of visitors which arrive through a search engine after entering certain keywords, using the *referrer* field of the log entry. Filtering can also be employed to remove duplicate entries due to visitors accessing multiple resources on the Web server. To avoid repeatedly looking up coordinates for the same IP address and sending these to the browser, the filtering component can be extended to skip requests from a known visitor. To do so, a list of recently encountered IP addresses is accumulated and only new addresses are forwarded to the next component in the pipeline.

7 Related Work

The most popular tool that shares the notion of pipeline with our approach is Yahoo! Pipes [4]. It provides a visual AJAX editor to interactively build pipelines for processing RSS feeds. Users can choose from a palette of predefined processing components. The result of a pipe is published as another RSS feed. The mashup application example presented in this paper could be built with Yahoo! Pipes only assuming that the Web server logs had been made accessible through HTTP and that a geocoding service was part of the palette. However, even with such a pipe, it would be difficult to provide real-time updates, as pipes have to be invoked in a request-response manner to retrieve their results. Thus, the user interface in the browser would have to poll the mashup's RSS feed periodically, which entails higher latencies and/or increased network usage.

A similar mashup application for mapping the location of upcoming calendar events has been featured in a Google Mashup Editor tutorial [21]. While the tutorial focuses on building a "pure" client-based mashup, the author also recognizes the benefits of a layered architecture, where an RSS feed with events is annotated with locations before it is forwarded to the mashup user interface.

As discussed in Section 4, pushing data from the Web server to the Web browser is not trivial. The communication between browser and server is constrained to the request-response pattern of HTTP. A paradigm shift towards *Comet* [23] has therefore been proposed, where Web applications use a bidirectional event-driven communication style. This paradigm allows both the browser and the server to initiate the transfer of data. Based on the Comet paradigm, Bayeux [24] is a protocol for client-server communication using the publish/subscribe model. Although it is based on HTTP, the protocol helps to overcome the request-response pattern to allow low-latency asynchronous communication between the client and the server. Since our mashup engine can be extended with new data ports, the Bayeux protocol could also be integrated in our mashup architecture to transfer the coordinates to the Web browser.

8 Conclusion

In this paper we advocate a layered approach to mashup design separating the integration logic from the presentation logic of the mashup Web application. This is mainly dictated by constraints that limit the applicability of the Web browser as an integration platform. Independently of this constraint, the benefits of layering can be understood in the context of the example mashup application presented in this paper. Given the dynamics of the underlying components, we argue that a layered approach to mashup design can also help to better deal with the evolution and maintenance of the mashup. This way, the mashup presentation logic is not affected by changes in the integration logic. Moreover, the pipeline defining the mashup integration logic should only be locally affected by changes in the mashed up APIs and data sources. The composition abstractions we have presented in this paper also help to deal with change in the technology landscape of the mashup environment. With our approach, the mashup logic is decoupled from the actual data access and service invocation mechanisms. Thus, the same mashup pipeline can run using traditional Web service protocols (SOAP), newer Web service protocols (REST/RSS), but also the most appropriate protocols for efficiently solving the problem (i.e., SSH and JDBC).

Acknowledgements

The authors would like to thank Thomas Heinis and the anonymous reviewers for their valuable feedback. Part of this work is funded by the European IST-FP6-15964 project AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

References

1. Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., Casati, F.: Understanding UI integration: A survey of problems, technologies, and opportunities. IEEE Internet Computing 11(3), 59–66 (2007)
2. Wikipedia: Mashup (web application hybrid), http://en.wikipedia.org/wiki/Mashup_web_application_hybrid
3. Abowd, G.D., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. ACM Trans. Softw. Eng. Methodol. 4(4), 319–364 (1995)
4. Yahoo! Inc.: Pipes, http://pipes.yahoo.com/
5. Corporation, M.: Popfly, http://www.popfly.ms/
6. Flake Media: GVisit, http://www.gvisit.com/
7. Bennett, K.: Legacy Systems: Coping with Success. IEEE Softw. 12(1), 19–23 (1995)
8. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Sebastopol (2007)
9. hostip.info: hostip.info Website, http://www.hostip.info/
10. MaxMind LLC: GeoIP, http://www.maxmind.com/app/ip-location
11. FraudLabs.com: IP2Location, http://www.fraudlabs.com/ip2location.aspx
12. IPligence Ltd: IPligence, http://www.ipligence.com/
13. Quova: GeoPoint, http://www.quova.com/
14. IP2Location.com: IP2Location, http://www.ip2location.com/
15. Asleson, R., Schutta, N.T.: Foundations of Ajax. APress (2005)
16. Biörnstad, B., Pautasso, C., Alonso, G.: Control the Flow: How to Safely Compose Streaming Services into Business Processes. In: Proc. of the 2006 IEEE Int. Conf. on Services Computing (SCC 2006), Chicago, USA (September 2006)
17. Internet Engineering Task Force: Extensible Messaging and Presence Protocol (XMPP): Core, http://ietfreport.isoc.org/rfc/rfc3920.txt
18. Mockapetris, P.: Domain Names - Concepts and Facilities. ISI (November 1987), http://www.ietf.org/rfc/rfc1034.txt
19. Crockford, D.: JSON: The fat-free alternative to XML. In: Proc. of XML 2006, Boston, USA (December 2006), http://www.json.org/fatfree.html
20. Pautasso, C.: JOpera: Process support for more than Web services, http://www.jopera.org
21. Hohpe, G.: Mashing it up with the Google Mashup Editor (June 2007), http://code.google.com/support/bin/answer.py?answer=71042&topic=12044
22. Eclipse Foundation: Eclipse Web Tools Platform, http://www.eclipse.org/webtools/main.php
23. Russel, A.: Comet: Low Latency Data for the Browser (March 2006), http://alex.dojotoolkit.org/?p=545
24. Russel, A., Davis, D., Wilkins, G., Nesbitt, M.: Bayeux Protocol – Bayeux 1.0draft 0. Dojo Foundation (2007), http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html