

## Chapter 7

# Advanced Mashups

**Abstract** In the previous chapter, we discussed the *internal architecture* and *integration* aspects of mashups, distinguishing between data, logic, UI, and hybrid mashups in function of where integration occurs in the application stack. In this chapter, we take a more *applicative* and *special-purpose* perspective on mashups and discuss some prominent examples of more advanced, focused types of mashups. Specifically, we look into *multiuser mashups* (such as *process mashups*), *mobile mashups*, *telco mashups*, and what is commonly known as (or our interpretation of) *enterprise mashups*. Technically, all these types of mashups build on top of the basic types we introduced in the previous chapter and can be seen as extensions thereof. The goal of this chapter is to illustrate how mashups can be tailored to specific application requirements or execution environments, adding additional value to the mere technical capability of integrating various resources accessible over the Web.

### 7.1 Introduction

Since their emergence several years ago as relatively simple, hand-crafted applications, mashups have been the object of investigation by researchers and practitioners coming from a variety of different backgrounds that, of course, also pursued very different goals with their investigation. For instance, today we also have mashups that enable new ways of communication to their users, or that aim at enabling the collaboration and/or cooperation of multiple users, both in simple Web contexts as well as in more demanding enterprise computing environments. These kinds of mashups can also be rather articulated pieces of software, whose integration logic goes much beyond the mere reuse of existing components and features.

In this chapter, we overview a set of representative examples of what we call *advanced mashups*. Specifically, we have a look at:

- **Multiuser mashups:** The mashups analyzed in the previous chapter were like conventional Web applications, that is, instantiated independently for each individual user of the mashup. Of course, there could be some exchange of

data via components that share data among multiple users, but this is rather an exception than the norm and not a form of multiuser support directly implemented by the mashup itself. Multiuser mashups, instead, are mashups that bring together multiple users in one and a same instance of a mashup and provide different levels of collaboration or cooperation. We distinguish three main types of multiuser mashups: mashups with *concurrent mashup components*, *concurrent mashups*, and *process mashups*.

- **Mobile mashups:** In line with the general trend in software/Web engineering and the growing demand coming from a user basis that is increasingly accessing the Web via mobile devices, mobile mashups aim at bringing mashups to mobile devices, such as mobile phones or tablets. In some cases, there is only the mashup as final application that is executed on the mobile device; in other cases, the mashup development environment may also run on the mobile device, further complicating the development of the mashup environment. Especially with the advent of the various device APIs (as we already discussed in Chap. 5), mobile mashups come with some interesting peculiarities.
- **Telco mashups:** Bringing together the power of both multiuser and mobile mashups, telco mashups are mashups that aim to provide people with novel, integrated communication capabilities and features. Both real-time (synchronous) and non-real-time (asynchronous) communications are possible. Rather than aiming at the communication among software components, telco mashups aim at the communication among people, for example, to further enhance collaboration. The peculiar characteristic of telco mashups is that they may cross the boundaries between the Internet and other telco networks (e.g., to send text messages or establish phone calls), creating a so-called converged network for communications.
- **Enterprise mashups:** Finally, enterprise mashups are mashups that usually run inside enterprise boundaries. While all the above-mentioned mashups and the mashups introduced in the previous chapter focus on the satisfaction of functional requirements, if mashups are to be executed in an enterprise environment some nonfunctional requirements also become important. For example, if a mashup is to be used to conduct business or to manage mission-critical situations, security and reliability become of utmost importance. In general, mashup users don't expect mashups to perform as good as regular Web applications or to be always available; in an enterprise context this is different, which requires some specific design support we will briefly summarize.

Next to these types of advanced mashups, there are of course many other possible configurations that may make sense. For instance, we could think of mashups for scientific workflows or other specific technological and/or application domains. We, however, believe the types of mashups discussed in this chapter are the ones most interesting today.

## 7.2 Multiuser Mashups

We start our investigation with *multiuser mashups*, which add a new dimension to the basic mashup design space already discussed in the previous chapter, that is, user management. In this context, a ***multiuser mashup*** is a mashup that brings together two or more users in a same *instance* of the mashup, so as to enable collaboration and/or cooperation among the users.

As the definition highlights, the key feature of multiuser mashups is that they bring people together. They mediate among two or more people, providing them with a dedicated collaboration and communication environment. Doing so means enabling people to work together on a same *instance* of the mashup at runtime, for example, to exchange messages, synchronize visualized content with each other, or even talk to each other. Typically, but not mandatorily, multiuser mashups are mashups with UI (either pure UI mashups or hybrid mashups), so as to enable user interactions.

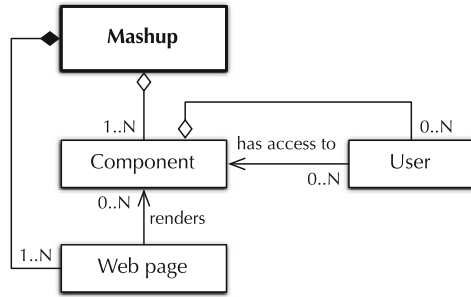
We distinguish three core types of multiuser mashups, which may also be used conjunctively in a same mashup, but we prefer explaining them independently, so as to highlight the respective peculiarities: mashups with *concurrent mashup components*, *concurrent mashups*, and *process mashups*. Although fundamentally different, these three types of mashups all share the common need to explicitly *manage user identities*, for example, to be able to correlate users with mashup instances at runtime. We discuss each of them next in increasing order of support required from the mashup logic.

### 7.2.1 Concurrent Mashup Components

The simplest configuration to bring together multiple users inside a same mashup instance is represented by mashups that make use of what we call *concurrent components*. A mashup with ***concurrent mashup components*** is therefore a multiuser mashup in which the collaboration/communication of multiple users is enabled by the components of the mashup, instead of by the mashup itself.

As the definition explicitly states, this kind of multiuser mashup does not require any specific support from the mashup logic. The key to enable multiple users to collaborate or communicate with each other is the use of components that allow them to do so. For instance, we can easily imagine a mashup that integrates a Google Doc or a Google Spreadsheet or a video conferencing widget into a common UI or hybrid mashup. If we use a Google Doc that is shared with a set of other people in read/write mode, once started in their respective Web browsers, the mashup allows these people to collaboratively and concurrently edit the document. Other possible components integrated into the mashup have local effects only. For example, next to the Google Doc component there could also be a calendar component, which allows each user to view his/her own appointments independently of the other users working on the Google Doc.

**Fig. 7.1** A simplified model of concurrent mashup components with components managing own users

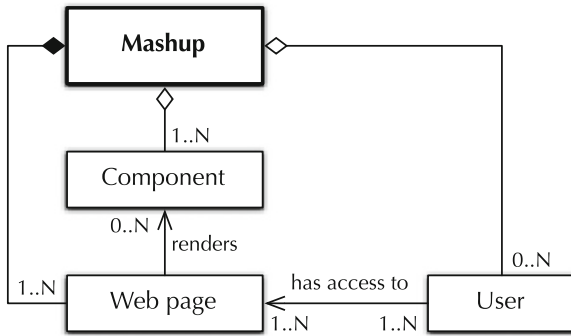


This is a very simple way of enabling multiple users to collaborate. In fact, as modeled in Fig. 7.1, all the user management is taken over by Google, which is in charge of managing user profiles, log-in, authentication, access rights management, data sharing, event propagation, and persistent data storage (of the document). The problems with this kind of mashups are essentially twofold: (1) it is hard to find many good components on the Web that internally already support this kind of user management and (2) if such components exist, they work in an isolated fashion, that is, they are typically not able to exchange runtime information with other components, independently of whether they are local single-user components or remote multi user components. Jointly using a Google Doc and a Google Spreadsheet would probably still lead to acceptable user experience (users and access rights are managed by Google in both cases), but if we mix, for instance, a Google Doc with a Skype conferencing component, it is obvious that both have, for example, different user IDs, policies and protocols. Getting them to work together would, of course, be technically feasible (e.g., by suitably wrapping them and extracting content), but it would surely be a very hard and error-prone endeavor.

Heinrich et al. [140] acknowledge this latter point and propose a generic collaboration infrastructure able to turn single-user Web applications into multiuser applications (they specifically focus on editors) by monitoring the DOM and propagating DOM modifications among participating parties. In [139], the authors apply their approach to the development of multiuser widgets, which can be mashed up (e.g., in widget runtimes like Apache Rave) and used collaboratively by multiple users, while in [141] they provide an annotation format that allows developers to natively extend their applications with collaboration support.

## 7.2.2 Concurrent Mashups

A way to solve the isolation problem of concurrent mashup components is to make the mashup itself concurrent: A **concurrent mashup** is a multiuser mashup that enables its users to operate a same instance of the mashup via a same view (e.g., a set of pages) in parallel, that is, concurrently.



**Fig. 7.2** A simplified model of concurrent mashups with multiple users accessing in parallel a same instance of mashup

That is, in a concurrent mashup it is the mashup that takes over the management of users and component synchronization. As illustrated in Fig. 7.2, this poses new requirements to the mashup itself: now it is the mashup that must be able to keep track of its user basis, to manage possible access rights (e.g., remember which user has access to which mashup or page thereof), and—more importantly—to synchronize at runtime the views of the different users participating in a same mashup instance. Storing user profiles and managing user registrations and access rights is relatively simple. What is complicated is the *synchronization* at runtime of the Web pages, that is, of the components running inside the Web browsers of the different users. The effect we would like to achieve with concurrent mashups is, for example, that if one user performs a selection of an item in one component, also all other components connected to the same mashup instance see the selection in their own browser. This requires UI events to be propagated from the user performing the selection to all other users whose mashups are in “listening mode.”

Extending what we have seen for UI mashups in the last chapter (and in line with the infrastructure support proposed in [140]), concurrent mashups must therefore implement:

- A ***distributed eventing infrastructure***, which allows a mashup instance in one Web browser to notify other instances running in other Web browsers about selected user interactions. It is generally neither necessary nor good design to propagate all low-level events from one browser to the other (e.g., mouse moves or mouse clicks that do not change the state of any component, or similar); it suffices to communicate those events that are of interest to all participants to a mashup instance. Which events are “interesting” depends on the specific mashup, and it is up to the mashup developer to configure event propagation properly. Technically, events may be notified by the source browser to the mashup server and then dispatched to all other browsers running the same mashup instance, or

it is also possible to send direct, peer-to-peer notifications among browsers via HTML 5 Web Sockets (<http://www.websocket.org/>).

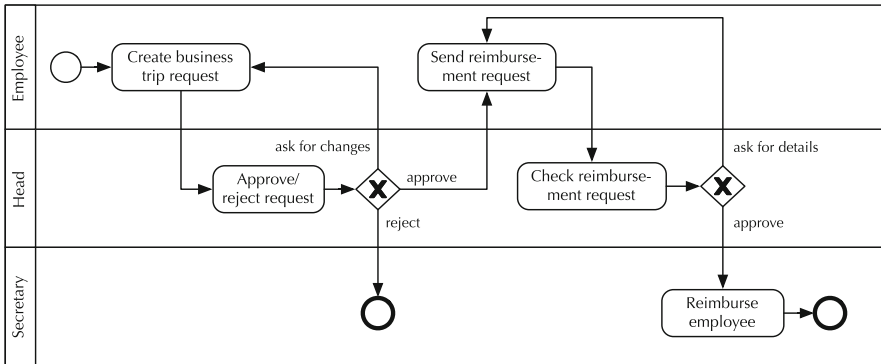
- A ***user-mashup correlation technique***, which allows the mashup to understand which user is participating in which mashup instance (consider that on a given server, there may run multiple instances in parallel, each with its own set of associated users) and the users to understand how to “connect” to an already running instance started by another user or how to invite other users to an own instance of the mashup. In the case of the Google Docs component, this correlation is, for instance, established via the name of the document worked on. Here, the mashup needs to be able to generate some form of runtime identifier (e.g., a *token*) that can be shared with the users that want to participate in a same instance. Technically, there are typically two options: either the user starting the first instance of the mashups invites other users to participate by sending them the token of the mashup instance, which they can then use to start their own instance of the mashup and to connect it to the one already running, or the mashup server may provide some form of dashboard that allows users to see which mashup instances that they have access to are currently running and to connect to them.

This latter option is, for instance, implemented in the MarcoFlow system [95, 97], which also uses a centrally mediated event propagation technique for the synchronization of remote components. MarcoFlow uses BPEL [161] to model its internal UI orchestration logic, and it requires the modeler to explicitly state which event of which component is to be sent to which operation of which other component. User management and correlation is taken over by the platform.

### 7.2.3 *Process Mashups*

The last level of complexity of multiuser mashups is represented by *process mashups*, which do not only bring together multiple users in a same mashup instance, by they also orchestrate the contributions/work of the individual participants. Weske [274] defines a ***business process*** as consisting of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations.

The definition introduces three new concepts: *activities*, which represent individual, atomic pieces of work that can be assigned to and performed by participants in the business process (e.g., select a book or pay an order); *coordination*, which structures activities so that their joint execution achieves a predefined effect (e.g., before paying the order, it is necessary to select the book); and the *business goal*, which is the effect the business process wants to achieve (e.g., online book sales). Usually, business processes are expressed via *business process models*, for example, using the Business Process Modeling Notation (BPMN [208]), which can be parsed



**Fig. 7.3** A simple BPMN diagram representing a business trip approval and reimbursement process involving three actors

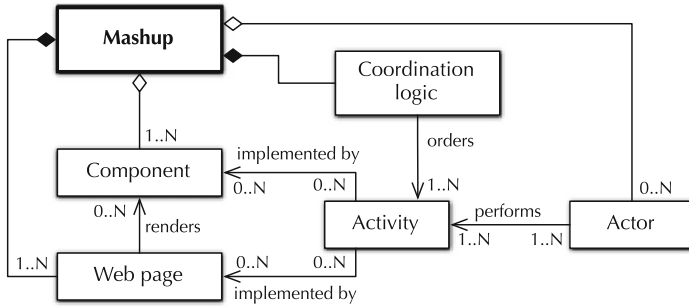
and enacted by a *business process engine*, causing the engine to interact with the people involved in the process and to automatically orchestrate their work.

Figure 7.3 provides an example of a business process model expressed in BPMN. The model illustrates a naive business trip approval and reimbursement process. The process is initiated by an employee, who creates a business trip request, which is inspected by the head of the employee’s business unit and approved or rejected; alternatively, the head may also ask for additional details or changes to the request. If the request is rejected, the process ends (represented by the bold circle). If the request is accepted, the employee goes on the trip (note that this activity is not represented in the model as it is out of the control of the business process engine) and, once back, submits his expenses for reimbursement. Again, the head has to sign off declared costs and, once approved, the secretary takes care of reimbursing the employee. Given this process model, a process engine can coordinate the tasks of the involved actors and automatically trigger them when their intervention is needed.

In process mashups, we do not necessarily have the same kind of separation of process model and process engine, and it is the mashup itself that acts as both process model *and* process engine. A **process mashup** is therefore a multiuser mashup that implements a business process by enabling its users to operate in parallel a same instance of the mashup via different views.

As illustrated in Fig. 7.4, process mashups, therefore, require the following additional ingredients, compared to the previous kinds of multiuser mashups:

- **Activities**, which represent the activities of the business process. A given activity may be implemented by one or more components or by one or more pages (in turn composed of components), depending on the complexity of the activity to be performed. For example, a simple “Approve business trip” activity probably requires only one component summarizing to the approving manager all necessary details of a trip request, while the more complex “Book trip” activity very likely involves multiple components, for example, for the selection of a



**Fig. 7.4** A simplified model of process mashups involving different actors performing activities

hotel, a flight booking component, a city map, local transportation maps, and similar; some of these latter components may be grouped into individual pages, so as to improve user experience. Activities are typically human activities, but some of them could also be mapped to Web service calls; in this case we speak about *automatic activities*.

- **Actors**, that is, the participants in the business process. Typically, there are two techniques to assign actors to activities: either the association is made directly inside the process model (in the mashup) or it is deferred to either deployment time or runtime by assigning only *roles* inside the process model and then resolving roles into specific actors later on. Common business process engines adopt this latter approach; for process mashups, the direct association of actors may also suffice, yet Torres et al. [262], for instance, propose the use of roles also for process mashups.
- A **coordination logic**, which structures activities into a business process. Individual activities may depend on each other, for example, the “Approve business trip” activity of course requires as input a request for a business trip by an employee. It is the coordination logic that assembles activities in such a way that (1) they are all provided with the necessary input data and (2) jointly they implement the expected business goal. This coordination logic is similar in nature to the integration logic we have seen for logic mashups, yet the focus here is on people, not on Web service calls.

The MarcoFlow system [95, 97] described earlier, for example, allows the implementation of process mashups. Its internal mashup logic is expressed via an extension of BPEL, that is, BPEL4UI, which enables the integration of UI components and Web services, the grouping of UI components into Web pages, the propagation of UI events among distributed instances of the mashup, the association of actors to pages, and the definition of the necessary coordination logic implementing the mashup’s process logic. At deployment time, a suitable model compiler parses the BPEL4UI definition and splits it into a standard BPEL definition, managing the invocation of Web services and the propagation of remote UI events, and a set of UI runtime configurations (similar to the ones used in



MashArt [93]), managing the rendering of UI components and the propagation of local, browser-internal UI events.

This is only one example of how process mashups can be implemented. In [94], the authors review different tools and their suitability for the development of process mashups. They specifically highlight the three core dimensions introduced by process mashups, that is, the multiple users, multiple pages, and business process logics, and argue that developing good process mashups is a complex task that only skilled developers are able to master.

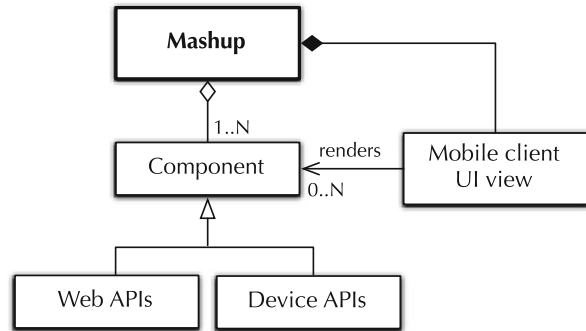
## 7.3 Mobile Mashups

Next to being used by multiple users, mashups are also increasingly becoming more personal and pervasive, in that they may also run on smartphones or tablets, that is, on mobile, carry-on devices. In this case, we speak about *mobile mashups*, that is, mashups that run on a mobile device (e.g., a smartphone or tablet) and that, besides integrating remote Web APIs, may make use of device-specific components, such as device APIs, and/or device-specific implementation frameworks.

As represented in Fig. 7.5, there are two peculiarities that characterize mobile mashups:

- **Device APIs:** Device APIs offer the users an enriched experience, in which device capabilities expand the common mashup features (e.g., integration of data and/or Web services) with auxiliary capabilities, such as dialing a phone number and establishing a phone call, saving a visualized event directly in the personal agenda, or accessing the GPS navigator to get the directions to a given address. Device APIs can, for example, also be used to manipulate integrated data sets, for example, displayed data in a UI view can be filtered based on the current user location made available by the device's location sensing module. Device APIs therefore provide the means for advanced hardware-software integration and seamless user experiences.
- **Mobile client UI view:** Since mobile mashups run on a client device, which have smaller screen sizes and resolution than desktop monitors, their UI must be structured differently than traditional UIs. For instance, while on a desktop monitor it is usually possible to visualize multiple UI widgets next to each other on a same page, this is typically not possible on a mobile device. Most of the times, on the mobile device there is only space for one UI widget at a time, and widgets must be distributed over more views (pages) than on desktop monitors. Then, mobile mashups may come in two different forms/implementations:
  - **Web-based mobile mashups:** The implementation of this type of mashups and their UI is based on standard Web technologies, and mashups are simply accessed via the mobile Web browser. Via the W3C standards for device APIs or the WAC proposals and the according browser extensions (see Sect. 5.3.4), Web-based mobile mashups have increasingly more access to local device

**Fig. 7.5** Simplified model of mobile mashups



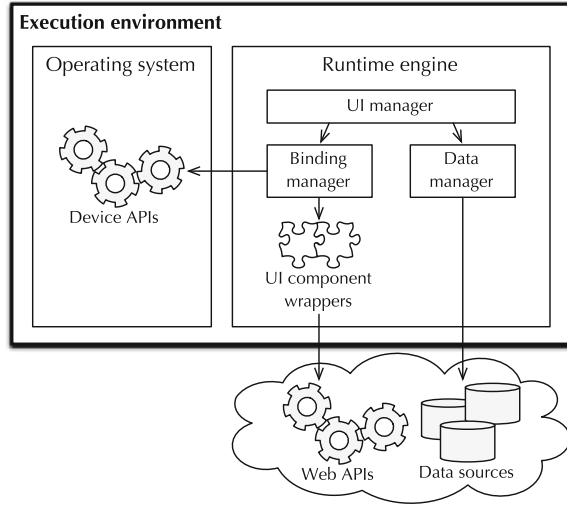
capabilities and can provide good user experience and integration with the device’s hardware capabilities. Similar to conventional Web applications, Web-based mobile mashups may run some client-side JavaScript application logic, yet that logic is subject to the JavaScript sandbox protection, which prevents access to generic Web resources. This requires the presence of a back-end Web server able to act as proxy and forward requests and answers.

- **Native mobile mashups (“mashapps”)**: The implementation of this type of mashups and their UI views is based on the native technology and programming languages of the client device; today’s most used mobile operating systems are Android, iOS, Windows Phone/RT, and Blackberry. UI views and the data they display can be synchronized with both Web APIs and device APIs, accessible via suitable operation system calls. Native mobile runtime environments typically also provide publish-subscribe communication channels for event-based communication, and UI components can be wrapped to comply with this event-driven logic. For example, it is possible to bind events of the UI views (e.g., the selection of a data item) with the invocation of operations in UI components (e.g., a search on Flickr, based on a search key selected in the core data view). Differently from Web-based mobile mashups, native mobile mashups are not sandboxed and therefore allow easy access to different remote data sources and services, which makes them less dependent on a Web server.

Both types of mobile mashups are typically *hybrid mashups*, where the integration of data sources produces *data views* that are visualized through *UI views* that in turn are able to synchronize with UI components and the device-native services through events—which is proper of UI mashups. Mobile mashups are also usually distributed over client and server, while native mobile mashups may also run on the client device only.

Figure 7.6 highlights the main architectural components in the runtime environment of a native mobile mashup [62]. The client-side logic consists of a *UI manager* handling the dynamic creation of UI views. It makes use of the device technology to generate the different “screens,” both the ones displaying the integrated data view

**Fig. 7.6** Execution of native mobile mashups on mobile devices in the MobiMash platform [60, 62]



and the ones for wrapped UI components. For example, in the Android operating system, the UI manager can be achieved through different *activities*, each one managing the generation at runtime of the code handling a specific screen.

The management of the data sets (service querying and result set parsing and manipulation) is then operated by a *data manager*. Typical choices need to be made regarding the integration of data:

- If a hosted solution is adopted (i.e., if there is also a server-side part of the mashup), data can be integrated at the server-side, based on a mashup configuration. At runtime, the data manager requests the integrated data set from the server, and the UI manager fills the UI views with the received results.
- Another option is to perform data integration at the client-side, in which case, due to the computing limitations of mobile devices, lightweight paradigms need to be devised. For example, the fusion of data coming from different services could be executed “on-demand,” that is, only when the user requests for specific data items for which fusion is needed [62]. In this architectural configuration, the data manager is in charge of querying the different data services and fusing/merging data items coming from the data sources, for example, based on their similarity.

A *binding manager* manages the coordination between the UI view and the components by exchanging messages with the mobile operating system. For example, Android *Intents* is a protocol supporting inter-app communication. Intents are abstract descriptions of operations to be performed, which can be used to communicate with any local component and background service. Thus, applications are enabled to request the execution of operations by any other application installed on the device, thus offering the possibility of implementing natively event-driven, publish-subscribe bindings.

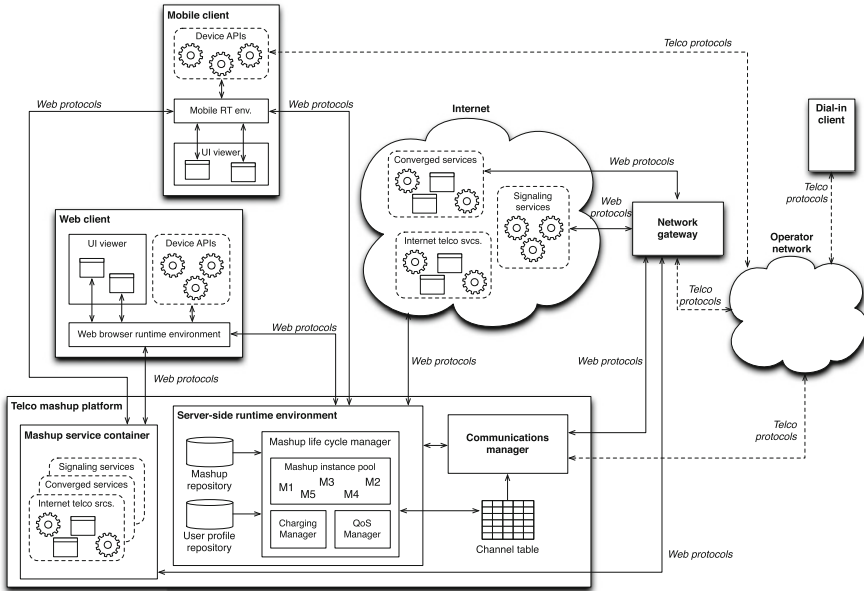


Fig. 7.7 Reference telco mashup architecture as defined by Gebhardt et al. [120]

## 7.4 Telco Mashups

The mashups that brings together the functionalities of both multiuser mashups and mobile mashups are *telco mashups*, that is, multiuser mashups that integrate telco services and/or device APIs to specifically support communication and collaboration among multiple users or to provide them with individual telco features [120].

Figure 7.7 depicts the telco mashup reference architecture introduced by Gebhardt et al. [120]. The figure has a particular focus on mashup platforms (see the *telco mashup platform* component), but nevertheless, the architecture illustrates well which elements may be involved in a telco mashup running on different devices (see the *Web client*, *mobile client*, and *dial-in client*). The core aspect of the architecture is that there are two networks that are different in nature, that is, the *Internet* and another *operator network* (e.g., GSM or UMTS for mobile phones or PSTN for traditional landlines). The Internet is based on the TPC/IP stack of protocols. Operator networks, depending on their nature, use different protocols (e.g., GSM or PSTN for traffic and SS7 or SIP for signaling). These protocols are incompatible with each other. This justifies the presence of the *network gateway*, which is the component that is able to mediate between different networks and, hence, different protocols. Network gateways are provided by the network operators and allow one to access operator network capabilities via the Web, for example, via suitable SOAP or RESTful Web services. For instance, a typical service exposed by such a network

gateway is one that allows one to send SMS messages from the Web to mobile phones. This is the most common way to cross the boundaries of the Internet for Web applications in general, yet, an application or mashup may also implement internally a *communications manager*, which is able to directly talk to the target operator network without the need for a network gateway. While this is theoretically possible, in practice the costs for this are prohibitive, and this is therefore not an option for individual mashups (the idea expressed in the figure is that a dedicated telco mashup platform could make this kind of service available to its mashups, that is, to a multitude of mashups).

Assuming a suitable communications manager is available, it is possible to implement, for instance, a mashup that connects users with the three client devices shown in Fig. 7.7: a conventional desktop Web client, a mobile smartphone accessing the Web via the Internet, and a conventional dial-in client (a phone) without visualization capabilities. In order to correlate the three devices, the communications manager would use an internal *channel table*, which allows it (1) to accept incoming phone calls and (2) to correlate them with the respective mashup instances running in the platform. In practice, this may happen as with today's phone conferencing systems: the initiator of the mashup notifies all participants about the dial-in phone number (to reach the communications manager) and the access code to be used (to correlate users with mashup instances). Next, the communications manager would mediate between the Web-based voice protocols used by the mobile and the Web client and the dial-in client.

However, today telco operators provide lots of Web-ready services through their network gateways, which makes the use of a dedicated communications manager unnecessary, and a whole mashup can be built by using special Web services. Examples of such services are the ones provided by Skype (<http://dev.skype.com/>), Voxeo Labs (<http://voxeolabs.com/>), or Deutsche Telekom (<https://www.developergarden.com/>), which enable the inclusion of advanced telco capabilities into common Web browsers.

It is this availability of telco services and APIs provided by the network operators themselves (such as Deutsche Telekom) that enables the development of complex telco mashups as well. While only few years ago these services and APIs did not exist and, therefore, the development of such kind of converged network applications required the availability of expensive communication managers, today all major telco operators provide Web-ready interfaces. Especially, the advent of voice-over-IP technology, as for instance adopted by Skype, has started eroding the traditional telco business based on voice/video calls and data services. Network operators are therefore looking for new business opportunities, and opening to the Web community is a natural move in this respect. Yet, the services and APIs provided through their network gateways still have one peculiarity that makes them fundamentally different from services as Skype: most of these services require some form of payment, for example, on a subscription or consumption basis. The reason for this is that they are not pure IP services; they are converged services that cross the boundaries of the Internet, respectively the operator network, and consume network capabilities (e.g., a communication channel) that have a cost for

the operator. Also, telco services may allow the configuration of custom quality of service characteristics, much more reliably than for pure Web-based services. Again, this has a cost.

As for the development of mashups, the challenge is the selection and seamless integration of Web-based and native telco capabilities that effectively serve user needs and do not overwhelm users with complexity. Voice and video calls are, for instance, synchronous communication channels with strict real-time requirements. This is fundamentally different from conventional Web services or Web-based components. Conciliating these needs with the typically less demanding (in terms of quality of service) needs of mashups is a challenge that needs to be addressed. The testing of telco mashups that integrate telco services may be complex because it may involve a cost; in most cases, however, dedicated developer accounts provide for enough free test capabilities. Finally, the billing of telco services in telco mashups or Web applications in general is a tricky endeavor, and some telco services may be available only in specific countries, which further increases complexity.

The EU FP7 research project OMELETTE (<http://www.ict-omelette.eu>) specifically focused on supporting the development of telco mashups by both developers and end users [76]. To the developer, it provides a Web service mashup environment (based on the mashup editor MyCocktail, <http://www.ict-romulus.eu/web/mycocktail>), which allows them to compose telco APIs into advanced telco W3C UI widgets as well as into long-running, server-side processes orchestrating telco APIs. To the end users, the project provides a simple widget mashup environment (based on Apache Rave), which allows them to integrate W3C widgets and to easily configure workspaces with advanced collaboration and communication features. The project comes with a rich library of readily usable telco widgets with features such as group voice calls, video conferencing, collaborative text editing, collaborative drawing, and similar. Workspaces can be shared among multiple users and turned into concurrent multiuser mashups. Integrated widgets are themselves concurrent mashup components. Interwidget communication is enabled via a suitable extension of W3C widgets with eventing capabilities and of Apache Rave with the necessary eventing infrastructure support. User authentication via WebID [252] for both workspaces and widgets enables single sign-on and seamless concurrency of both components and workspaces.

## 7.5 Enterprise Mashups

The last type of advanced mashups we would like to briefly discuss is enterprise mashups. There are lots of interpretations of what the addition “enterprise” means and of what the difference between enterprise mashups is compared to other types of mashups, and we did not find good definitions so far. According to our interpretation, the core characteristic differentiating enterprise mashups from others is their execution context, that is, the enterprise context: An *enterprise mashup* is a

mashup that serves a business purpose, runs in the context of an enterprise network, and may be subject to enterprise-specific, nonfunctional requirements.

That is, we do not identify any specific functional feature that is able to allow us to tell enterprise mashups and other types of mashups apart inside the complex mashup landscape. As a matter of fact, enterprise mashups could be of any of the types we have seen so far, ranging from simple widget-based UI mashups to collaborative, multiuser mashups and, of course, process mashups. What is important, however, is that an enterprise may have some *nonfunctional* requirements, such as security or compliance, that, if not met, may impede the use or hinder the applicability of mashups inside an enterprise.

As for the functional features, undoubtedly data integration, transformation and mediation from disparate information sources (especially if data are sourced from the Web and integrated with corporate data, for example, in the case of Web-based business intelligence) is one of the biggest challenges enterprises face in general and, therefore, it must also be addressed in the development of enterprise mashups. In Chap. 2, we have seen that data integration is generally a complex task; in Chap. 6, we have seen how simplified data integration can be achieved with mashups. Enterprise mashups may require a combination of both approaches to achieve robust, scalable, and consistent data integration in a flexible and component-based fashion. In addition, while *interwidget communication* is nowadays common practice in widget-based UI mashups, enterprise mashups may further require *inter-mashup communication*, for example, enabling two long-living mashup instances (processes) to exchange information and to synchronize on predefined guard points or to enable collaboration of multiple employees running instances of different mashups. This latter point is still in its infancy and requires more investigation, but it also exceeds the boundaries of the term “mashup” as we defined it in the introduction of this book (non-mission-critical, simple Web applications).

The key *nonfunctional requirements* we identify for enterprise mashups are (of course, each enterprise will have its own policies and standards):

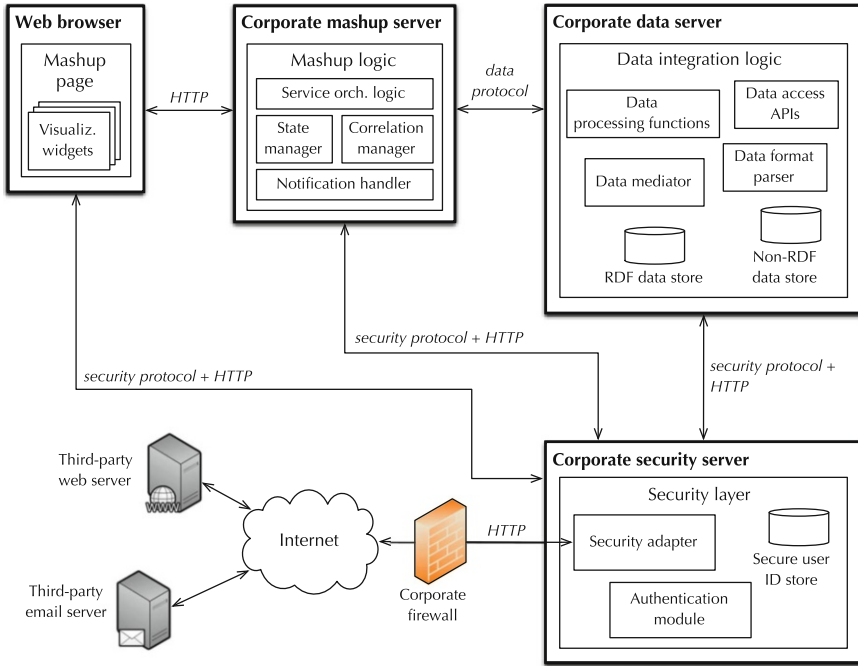
- **Use of standards:** Standards strictly focusing on mashup development have begun to emerge only recently and mature only slowly (e.g., EMMML [215]), but the use of standards in general may be an enterprise-internal policy to enhance the understandability, maintainability, and return on investment (ROI) of own software. Typical standards in this contest are XML, XHTML, RDF, OpenSocial, Microformats, OpenID, WebID, OAuth, WS-Security, and so on. Of course, ad hoc developed mashups for personal use are not subject to these kinds of restrictions and typically trade standards compliance for development speed.
- **Data quality:** Data is today probably the single most important asset of modern enterprises. Yet, data is not just data. Data, if not of suitable quality, can even be useless. With the continuously growing amount of data, more and more information is accessible in digital format and available to be processed by machines and visualized to users for decision making. It is of utmost importance that data used for decision making is of high quality, for example, in terms of

completeness, freshness, consistency, so as to support good decision making. This is not different for mashups [56].

- **Testability:** Since enterprise mashups may also act on corporate data and modify them, it is important to support ways to test mashups, for example, in a protected environment, before allowing them to affect the corporate data assets. Read-only mashups, for example, for information presentation, are less critical in this respect.
- **Availability and robustness:** Mashups are often considered simple applications for *situational* use only, that is, applications that allow one to answer a given question or query and that become obsolete once the answer is found. But mashups, especially in the enterprise context, may also be applications that add value to multiple people and that are of longer validity than just one answer. These kinds of mashups must be more “mature” and guarantee the necessary availability and robustness to its users, just like any other software product used by the enterprise.
- **Security and single sign-on:** This is probably the most important requirement an enterprise may pose to its mashups. Since enterprise mashups may present and act on corporate business data (also mashing them up with external data sourced from the Web), deciding who is allowed to access and run the mashup is a sensible and crucial aspect. Both the mashup and its components may require user identification and authentication. In order to provide for good user experience, single sign-on (the capability of the mashup to automatically authenticate a user with its components after the first log-in) becomes, therefore, almost a must.
- **Governance and compliance:** Finally, just like any other software product used by an enterprise, mashups must allow for their governance (from an IT perspective) and the assessment of their compliance with regulations, laws, policies, and similar (from a legal and business perspective). The former may be guaranteed via the use of suitable mashup platforms, the latter requires more elaborate analyses of the mashups, logging, user authentication, suitable protocols, and the like, to allow external auditors to assess and certify compliance—all aspects that are outside the scope of all types of mashups we discussed so far.

Figure 7.8 describes a simplified multitiered, distributed architecture for enterprise mashups. The architecture is based on the proposal by Hanson [135], which also proposes the use of semantic technologies for data integration (we consider this an optional feature) and adapted to the concepts and architectural elements introduced earlier in this book. The architecture has a clear separation of concerns: a security server is in charge of user identification and access right management; it also mediates all communications from the enterprise-internal network to the Internet and vice versa. All internal servers and applications/mashups communicate with the security server via dedicated security protocols for authentication (e.g., OAuth) and HTTP for payloads. The data server comprises the data mediators, data format parsers, data processing functions, and data access APIs. These latter





**Fig. 7.8** A typical enterprise mashup architecture with logic layer, data layer including semantic data repositories, and security layer (based on [135])

are accessed by the mashup server via suitable data protocols (e.g., SQL) or Web service calls. Service are orchestrated using a state manager, correlation manager and notification handlers. The actual mashup is visualized in the client browser via suitable visualization widgets.

One way to address the *testability* requirement introduced above is the use of so-called *innovation toolkits* [96], which can be seen like sandboxed mashup environments that allow employees to play with the information assets of the enterprise (typically data and processes) and to mash them up in new, hopefully value-adding manners. The innovation toolkit is provided and maintained by the enterprise's own IT department, which monitors the evolution of the mashup ecosystem in the sandboxed environment and may decide to "upgrade" some of the mashups, reimplementing them according to enterprise standards and making them available to the whole enterprise. We will come back to the idea of innovation toolkits in Sect. 9.2, where we discuss the power of innovation toolkits to enable end-user innovation.

## 7.6 Summary and Bibliographic Notes

In this chapter, we overviewed some of the mashup types we call “advanced,” that is, mashups that can be characterized not only in terms of where in the application stack the integration of components happens, but more in terms of what kind of applications or uses they support. The last chapter was more technology-centric and generic, this one is more application-centric and specific to application scenarios (e.g., collaboration or processes). Since the number of application domains is essentially unlimited, for example, compared to the number of technologies and the number of integration techniques that characterize mashups, this chapter is by no means intended as exhaustive or comprehensive. Many different types of mashups tailored to specific functionalities, purposes, and users can be envisioned, are already in use, or will emerge sooner or later. The goal we pursued with this chapter is to have a look beyond the mere practice of integration and to show some prominent examples of useful specifications of the generic types of mashups introduced in the previous chapter. We believe it is important to understand that developing good mashups is not just a matter of knowing technologies and of knowing how to hack into existing applications, it is also a matter of knowing the problem domain and of knowing how to support the problem domain’s requirements via mashup technology and practice.

We also think it is not possible to provide a list of further readings that covers all the possible configurations of mashups that are there. It is probably best that the interested reader stays informed about the mashup ecosystem via Web sites like <http://www.programmableweb.com> or via the major scientific conferences related to mashups and integration on the Web.

However, who is interested in deepening his/her understanding of process mashups, is referred to the work by Daniel et al. [94], which studies a set of different mashup tools and classifies them according to their capability to support process mashups. More fundamental knowledge on business process management in general can be found in [274], and it may be good to have a look at the BPMN [208] for all details on how to model business processes. As for telco mashups, we already provided a link to the OMELETTE project (<http://www.ict-omelette.eu>), which specifically focuses on telco mashups and also end users; Gebhardt et al. [120] provide a summary of the key aspects of the project. Heinrich et al. [138, 140, 141] provide interesting details on how to turn single-user Web applications or widgets into multiuser applications or widgets. And, finally, Hanson’s book *Mashups: Strategies for the Modern Enterprise* [135] treats very well all the different aspects and nonfunctional requirements that enterprise mashups should support; we only hinted at them in this chapter.

Mashups

Concepts, Models and Architectures

Daniel, F.; Matera, M.

2014, XIX, 319 p. 119 illus., 2 illus. in color., Hardcover

ISBN: 978-3-642-55048-5