Mashups have emerged as an innovative software trend that re-interprets existing Web building blocks and leverages the composition of individual components in novel, value-adding ways. Additional appeal also derives from their potential to turn non-programmers into developers. Daniel and Matera have written the first comprehensive reference work for mashups. They systematically cover the main concepts and techniques underlying mashup design and development, the synergies among the models involved at different levels of abstraction, and the way models materialize into composition paradigms and architectures of corresponding development tools. The book deliberately takes a balanced approach, combining a scientific perspective on the topic with an in-depth view on relevant technologies. To this end, the first part of the book introduces the theoretical and technological foundations for designing and developing mashups, as well as for designing tools that can aid mashup development. The second part then focuses more specifically on various aspects of mashups. It discusses a set of core component technologies, core approaches, and architectural patterns, with a particular emphasis on tool-aided mashup development exploiting model-driven architectures. Development processes for mashups are also discussed, and special attention is paid to composition paradigms for the end-user development of mashups and quality issues. Overall, the book is of interest to a wide range of readers. Students, lecturers, and researchers will find a comprehensive overview of core concepts and technological foundations for mashup implementation and composition. Even without low-level coding details, practitioners like software architects will find guidance on key implementation concepts, architectural patterns, and development tools and approaches. A related website provides additional teaching material which can be used either as part of a course or for self study.

This book is timely, provides a thorough scientific investigation and also has practical relevance in the general area of composition and mashups. It is of particular interest to researchers and professionals wishing to learn about relevant concepts and techniques in service mashups, composition, and end-user programming.

From the Preface by Boualem Benatallah, University of New South Wales, Sydney
The characterization of mashups as applications that introduce added value through component integration is not obvious. A study that we conducted on the large on-line repository of mashups published by programmableweb.com, showed, first of all, that there is no general consensus on what a mashup is and what it is not. The systematic analysis of a sample of about 150 different mashups randomly selected out of the whole repository using a Simple Random Sample (SRS) technique revealed that 29% of the considered mashups make use of only one single API (in most cases a map), without featuring an integration of multiple components into a new application. This means that many of the "mashups" published on the Web can actually not be called "component-based applications" or "composite applications." However, as our definition shows, we strongly believe that it is the presence of multiple components (at least two) and their sensible coupling that provides most of the added value that mashups can bring to their users. For instance, integrating a Google Map to graphically visualize an address, as many companies, restaurants, institutions, and similar do today (e.g., in their How to get there page) is just not enough to call a web application a mashup – at least not from our point of view.

In Figure 6.1 we roughly position the types of applications that we consider mashups, compared to the more traditional integration practices of application integration and data integration introduced in Chapter 2. The characterizing dimensions are the application complexity and the layer of the application stack at which integration is performed. One can immediately note the contribution of mashup development as practice enabling integration at the presentation layer and typically focus on non-mission-critical applications.

Fig. 6.1 Positioning of mashups compared to other integration practices, such as application integration and data integration. Mashups introduce integration at the presentation layer and typically focus on non-mission-critical applications.
Fig. 6.2 The basic mashup model: a mashup integrates a set of components, possibly puts them into communication, and optionally renders results or components.
Fig. 6.3  A conceptual model for data mashups.
From an implementation point of view, there are many ways to implement data mashups. The rest of this section describes in particular three architectural patterns that we consider the most representative ones; different variations and nuances thereof may of course exist.

It is important to note that the model we propose in Figure 6.3 illustrates the characteristics of the most used types of data mashups so far, i.e., stateless data mashups that are processed in one shot and that terminate after producing an output. We however acknowledge that we may also have streaming data mashups (e.g., [43]), which are of long-living nature in that they stay alive and process data items (e.g., sensor readings from a wireless sensor network or new items added to an RSS feed) until the mashup is not explicitly terminated. We defer the discussion of how the necessary interaction state with the streaming data source can be managed to the explanation of logic mashups; the rest of the internal logic of streaming data mashups is as explained in this section.

6.3.1 Point-to-point data mashups
The first architectural pattern we call point-to-point data mashups, as data integration is achieved as the result of a direct interplay of data sources with data processing functions or of one data processing function with another, with the mashup establishing the necessary direct point-to-point communications.

Fig. 6.4 Basic data mashup architecture with direct data passing among data processing functions.
Fig. 6.5 Data mashup architecture with data mediation and integrated data store.

In point-to-point data mashups, where there is no data mediation, each data processing function has to understand two potentially different data models, i.e., the data model of the input data and that of the data produced as output.

Given the similarity of the architectures of the two types of data mashups, also their mashup characteristics are essentially the same. Yet, in centrally mediated data mashups, we can no longer have direct data passing among source components and data processing functions or among functions themselves. Data passing from source components to the integrated data store is mediated, data passing among data processing functions is typically based on a shared memory (the integrated data store).

6.3.3 Data mashups with external data processing logic

Finally, the last type of data mashups we consider are data mashups with external data processing logic (see Figure 6.6), which, besides internal data processing functions, make use of web services or similar to reuse third-party data processing capabilities and power. Not always it is possible to get access to all the necessary data sources to obtain a desired output (e.g., by joining the data), but suitable web services may help out. For instance, if we want to add human-understandable location information to an RSS feed containing GPS coordinates (expressed via longitude and latitude geo-coordinates), it is
practically impossible to find a data sources for all cities and street names that could be joined with geo-coordinates. However, there exist, for example, free services accessible over the Web that provide for the translation of geo-coordinates into city and street names, which can be used for this purpose. The integration of these kinds of services serves the purpose of transforming data and should not be interpreted as web service composition, as described in the next section. The characteristics of data mashups with external data processing logic are therefore the same as the ones of any of the two previous data mashup configurations, depending on whether the mashup is based on centrally mediated data or not. Of course, unlike internal data processing functions, external data processing services cannot access the integrated data store directly and therefore require data passing by value.

6.4 Logic Mashups

Logic mashups extend the scope of integration from data to logic components. A logic mashup integrates mashup components at the application logic layer of the application stack, by enabling the composition of functionality.

Fig. 6.6 Architecture of a data mashup with external data processing logic.
In addition to managing the various aspects of data integration as discussed for data mashups, integrating logic components (e.g., web services) specifically asks for the orchestration of the respective communications with components. Data mashups essentially neglect this aspect, since data components typically act as static data resources and do not accept data as input, nor do they process data on behalf of their clients. This is the core feature provided by logic components, which, given the different ways remote application logic may be delivered to its clients, however come with some peculiar requirements that a developer must master when developing logic mashups.

Figure 6.7 illustrates an according conceptual model that extends that of data mashups with five new concepts (next to highlighting that now components may have multiple operations):

- **Synchronous communication**: This is the most common invocation paradigm for logic components, not only on the Web. As already introduced in Section 5.3 when discussing the different types of logic components, synchronous communication may refer to local or remote procedure calls and are blocking communications. Local procedure calls are used,
6.4 Logic Mashups 165

Web server

Logic mashup

Public service API

Mashup control logic

Data processing functions

Data format parser

Data mediator

State manager

Correlation manager

Notification handler

Protocol adapter

SOAP service  RESTful service  Web page

Fig. 6.8 Generic architecture of logic mashups with support for synchronous and asynchronous communication; data are processed like for data mashups.
Fig. 6.9 User interface mashup model without inter-component communication.

The model comes with two extensions, one regarding the rendering of components and one regarding the layout of the components inside the mashup's UI. The extensions refer to the following elements:

- **User interface**: UI components have a native UI that can be reused as is for the development of the mashup's composite UI. Reusing a component's UI generally enables users to interact with the component in the most appropriate fashion and relieves the mashup developer from one of the most sensible aspects of software development, i.e., interaction design.

- **Templates**: the layout and style of the overall presentation of the mashup. UI mashups have a user interface that partly derives from the composition of component UIs, partly from the hosting template that hosts the components and adds additional style and content elements to them (e.g., suitable titles or background images). As the rendering of the mashup occurs in the web browser, templates typically come in form of HTML pages, each page able to host one or more UI components.
meaningful state changes, which may be of interest to other components of a some mashup, not mere JavaScript or DOM events (like mouse moves or clicks). UI events, like any other operation, may carry parameters with them, for example to communicate which new city has been selected in a map component; the data entity in the model also represents event parameters.

• Shared memory: Inter-component communication can make use of a shared memory for the exchange of data among components, e.g., in the absence of UI events or when data are too big to be passed via events. Components may also use a shared memory to store data and use events to notify other components about the availability of new data. In principle, a shared memory also enables data mediation (to transform data), but UI synchronization typically requires exchanging only simple data structures or even scalars (e.g., parameter-value pairs), which do not require complex transformation capabilities.

In the rest of this section, we illustrate different ways to achieve integration of UI components in practice. As we have seen, UI integration is intrinsically event-based, a property that distinguishes it from integration at the data or logic layers. However, given the relative immature technologies involved in UI integration and their heterogeneity, it is common to find a mix of different communication and integration techniques at the presentation layer. Also in the case of UI mashups we may have mashups that integrate streaming components, e.g., a multi-media component that allows users to

---

Fig. 6.10 User interface mashup model with inter-component communication.
Fig. 6.11 The simplest UI mashup: embedding external resources inside own HTML code.
applications. The mashup consists of an HTML page, which may include URI-based references to entire Web pages, multimedia content or JavaScript UI libraries. When rendering the page, it is the web browser who requests linked sources and makes them available inside the mashup. Resources embedded via iframes are isolated and executed as if rendered in an own web browser. This architecture of UI mashups clearly highlights the UI-based integration logic with no data passing among components. The presentation logic is based on reused UIs (the iframes, multimedia objects, or images), and the instantiation model is by definition short-living.

6.5.2 Wrapped UI mashups

We now try to understand how to obtain UI mashups that, with some additional logic to instantiate and invoke components, also support the synchronization of UI components during the mashup execution (see Figure 6.10 for the respective mashup model). Given UI components like HTML snippets, third-party web sites or similar, which were not developed for interoperability, achieving synchronization typically requires two new ingredients: wrappers, i.e., extensions of the basic component models (e.g., client-side scripts), which are able to provide the components with a suitable UI and/or to equip them with the necessary support for inter-component communication; and an integration logic that puts the components into communication.

Let's consider the development of a mashup that helps its users to find music events in the Milan area using a list of concerts, a map, and a calendar. The involved components are a concert listing providing a list of music events as an RSS feed, a map able to render points of interest, and a calendar view of the concert list. The combined effect that we want to achieve is that when
Assembling a portal page is performed in two phases: (i) generation of markup fragments by portlets and (ii) aggregation of fragments into the portal page. The portlet markup fragments adhere to rules that facilitate content aggregation into portals. Portals interpret the portlet markup code, allocate suitable space for the rendering of each portlet, and generate the composite UI. Portals typically allow users to customize the composite UI (e.g., to rearrange or show/hide individual portlets), and provide facilities like single sign-on and role-based personalization for portlets.

Analogous to Java servlets, portlets implement a standard Java interface (JSR-168 [1]), to enable developers to create portlets that can be plugged into any standard-conform portal. JSR-168 also defines a runtime environment for portlets, the portlet container, and the Java API between the container and the portlets. Figure 6.13 illustrates a typical portal architecture. The portlet container hosts portlets and operators support for their deployment and execution, i.e., it provides the required runtime environment, e.g., with support for persistent storage to store portlet preferences. The portlet container receives requests for the execution of portlets from the portal, where the actual user interaction with the portlet takes place. As such, the portal aggregates the markup of its portlets and manages communications with the portlet container in a centrally mediated fashion. That is, the portlet container is not responsible for aggregating and displaying the fragments produced by the portlets; this is under the responsibility of the portal.

JSR-168 focused on portals that use only portlets installed locally in the portlet container. The Web Services for Remote Portlets specification [2] then standardized the interaction with remote portlets accessed via SOAP.
Fig. 6.14 An example of widget-based mashup created using the Netvibes platform. Different areas of the page correspond to different viewports, each one displaying the content of a different widget.
However, the technology is still young and evolving. There are indeed already research works (e.g., [252], [277]) that have started proposing extensions to the widget model to make widgets inter-operable (at least within a same page). Similar discussions are also ongoing in the widget standardization group. The most accredited approach proposes extending the W3C widget model with client-side event generation and handling capabilities. For example, the approach presented in [277] proposes the addition of a dedicated Intercom Interface that extends the W3C Widget Interface to support: (i) raising events, i.e., producing messages to communicate internal state changes, (ii) invoking operations on widgets, and (iii) exposing metadata about the events and operations supported by a widget. Event transmission is mediated by a dedicated client-side event bus (see Figure 6.15 or publish-subscribe frameworks (e.g., pmrpc, http://code.google.com/p/pmrpc/) as extensions of the widget runtime environment. Chudnovskyy et al. [76], instead, describe a technique to wrap widgets and to equip them with event handling support, if they don't support events natively.

These extensions enable both the orchestration and choreography of widgets within a same page: In an orchestrated widget integration, it is the central mashup logic that subscribes operations to events, as for example shown in [277]. In a choreographed widget integration, each widget publishes its events...
Fig. 6.16 The hybrid mashup model conciliating integration at the data, logic, and presentation layer.
For each of these types of hybrid mashups, we can then have different types of architectures, especially depending on how integration logic is distributed over the different tiers of the (distributed) mashup. For simplicity (and as this is the most common case), in the following we specifically focus only on one client tier and one server tier and discuss three typical examples of hybrid mashups and illustrate possible architectures.

Figure 6.17 shows an example of data/UI mashup with client-side data integration and presentation logic and with an intermediate data interpretation step executed at the server side. In particular, the architecture proposes a configuration where the mashup is started and operated by the web user via a common web browser, while the back-end part of the mashup running on the web server serves as proxy toward the actual data sources and to pre-process fetched data and transform them into a common format (e.g., JSON) that is immediately accessible to the client-side data integration logic (e.g., coded in JavaScript). Data integration at the client-side is not a typical...
Fig. 6.18 Architecture of a data mashup with client-side rendering of results and possible intermediate progression information.
Web browser

Web server

Fig. 6.19 A possible architecture for a universal mashup, integrating web services and processing data on the server side while using custom UI widgets for the visualization of results and the interactive control of the mashup.
Mashups are a new type of integration, fueled by the constantly growing availability of reusable resources on the Web. In the previous chapter, we have seen that the Web offers a very rich set of reusable resources, which may turn mashup development into a complex endeavor. The chapter is structured according to the four values of the first mashup characteristic, the mashup type. Considering the possible values of the other characteristics gives an idea of the variety of the possible implementations of these basic types of mashups.

One interesting aspect of mashups, as applications focusing on the reuse of web resources, which we did not consider throughout our discussion, is the reusability of mashups themselves as components for other mashups. Alternatively, we can also speak about hierarchical composition. While for data and logic mashups this is not an issue, as they typically are delivered via RSS/Atom feeds or web services that are reusable components by definition, reusability becomes an issue for UI and hybrid mashups. Of course, it is always possible to extract pieces of UI from any kind of web application, but this is generally not a good practice, and one would simply expect more attention to this problem from mashups. However, as we will see when discussing mashup platforms, the publication of mashups as reusable components is mostly neglected or hard to achieve with current implementations, e.g., based on the runtime interpretation of mashup model instances (diagrams).

As for the further reading, we already discussed data and application integration in Chapter 2, introducing the basics and foundations underlying both data and logic mashups. Doan et al. [107] provide a very good summary of the problems and solutions regarding data integration. Alonso et al. [13] surely represent a reference for application integration with a special focus on the case of web services, while Papazoglou [224] more specifically explains the use of web service technologies in practice, e.g., also providing good insight into the problem of correlation in the context of BPEL. Although mashup do not aim at the full power of BPEL, the problem re-presents itself in similar terms.

Fig. 6.20 Seven characteristics to distinguish different mashup models.