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
However, very often mashups also adopt a server-side logic. In some cases, the server is exploited only for the storage of the resources (e.g., the HTML and JavaScript files) and application data, while the application logic runs at the client side and communicates back with the server only for operations on the persistent data. In some other cases, the server also executes (part of) the application logic. Several data mashups, for example, make use of complex queries on multiple data sources that require a computation that would not be affordable within the Web browser. Data integration is therefore operated at the server-side. Process mashups also make use of a server-side logic to combine functionality into one or more external processes. Given these scenarios, this chapter aims to give an overview of the most prominent technologies for the development of mashups executed on the Web.

We start with a short refresh of the Internet, its structure and underlying protocols, so as to then illustrate the technologies for the development of Web applications on top of this infrastructure. We discuss the most prominent languages for client-side and server-side programming and the main formats for data representation on the Web. Our aim is neither to be omni-comprehensive nor to teach how to use specific technologies; rather we want to illustrate the most prominent representatives of the panorama of technology choices that can be made when developing Web mashups.

**Fig. 3.1** The change of the distribution of a web application’s internal architectural layers over client and server over time (adapted from [http://www.coachwei.com](http://www.coachwei.com)).
Fig. 3.2 Internet architecture (adapted from [259]). The dashed polygon describes a possible distribution of a web application over the Internet.
The bottom of the stack is represented by the host-to-network layer, which corresponds to the OSI datalink and physical layers. This layer is concerned with the physical encoding and transmission of bits over the network and not of great interest any longer today (by now, Ethernet has almost become a de-facto standard). The actual contribution of the TCP/IP stack, as its name already tells, starts with the internet layer. We intentionally do not use an uppercase for this layer, in order to emphasize its actual purpose, i.e., inter-connection different networks. What we know as the Internet (with the uppercase) is a particular instance of network implementing the TCP/IP stack that takes its name from the enabling OSI network layer. The internet layer is in charge of delivering IP packets from one host to another. It does not guarantee for the order of packets; the major issue is routing. It is the transport layer on top that is in charge of ordering and assembling IP packets. Two transport protocols exist: TCP (Transmission Control Protocol) is a reliable, connection-oriented protocol; UDP (User Datagram Protocol) is an unreliable, connection-less protocol. The former is typically used for applications, the latter, for instance, for media streams (where occasional packet losses are acceptable). On top of these two layers, we already have the application layer. The TCP/IP stack does not provide a presentation or session layer like the OSI model; these two aspects can be managed by the applications themselves. But we have different protocols at the application layer: TELNET, FTP, SMTP, NNTP, HTTP, DNS, and many more.

Up to the early 1990s, the Internet provided four main types of applications: email (SMTP), news (NNTP), remote login (TELNET), and file transfer (FTP). Only in 1989, the physicist Tim Berners-Lee invented the World Wide Web based on the Hypertext Transfer Protocol (HTTP).

Fig. 3.3 The TCP/IP reference model compared to the OSI reference model.
interprets the meaning of the tags and transforms the marked up content. This transformation is then managed by a processor embedded in the Web browser, which is responsible for presenting such content or to define links to other Web pages. The document presents a simple Web page linking to the Expo2015 Web site.

![A simple Web page linking to the Expo2015 Web site](Logo_EXPO.png)

**Fig. 3.4** A simple HTML page embedding an image and a video and including a clickable hyperlink.
mask interoperability problems and also facilitate the use of native JavaScript development frameworks have been proposed to cope with interoperability problems, and also to facilitate programming, some execution of a same code might not result into the same page behaviour. To through the array, element by element, every time the link is clicked.

The function in particular defines an array of videos and manages moving through the array, element by element, every time the link is clicked.

```html
<html>
<head>
<title>EXPO 2015</title>
<script type="text/javascript">
    var nextButton;
    var videoCanvas;
    var videoArray;
    var currentVideo = -1;

    function nextVideo() {
        currentVideo++;
        if(currentVideo >= videoArray.length)
            currentVideo = 0;
        videoCanvas.setAttribute("src", videoArray[currentVideo]);
    }

    function init() {
        nextButton = document.getElementById("next_button");
        videoCanvas = document.getElementById("video_canvas");
        videoArray = [
            "http://www.youtube.com/embed/kNG_1UKkgM",
            "http://www.youtube.com/embed/JdKlb1g1VvA",
            "http://www.youtube.com/embed/m_F8A5VdhM"
        ];
        nextButton.addEventListener("click", nextVideo, false);
        nextVideo();
    }
</script>
</head>
<body onload="init()">
    /* mark-up for other page elements */
    <p>&lt;a id="next_button" href="javascript:nextVideo()">Next video</a>&lt;/p>
    &lt;iframe id="video_canvas" width="800" height="500" src="" frameborder="0"
allowfullscreen&gt;&lt;/iframe&gt;
</body>
</html>
```

Fig. 3.5 An example of JavaScript code included in the head section of an HTML page. The script adds interactivity to the page, by implementing a video slide show.
Web Technologies

Some input parameters and generates an output page. The invocation of the external program occurs when the HTTP request includes a URL pointing to a program instead of to a static document. To make the server communicate with the external program, one of the first conceived solutions was to use a standard interface, called **Common Gateway Interface (CGI)**, to allow the Web server to invoke **CGI programs**, executed to dynamically construct the page. A CGI program can issue queries over a database to extract data which it then uses to assemble an HTML page, or it can store user inputs in the database by inserting or updating data in it.

The CGI architecture, however, soon showed some limitations, first of all its performance, which is downgraded especially by the strategy for managing processes. For each HTTP request for a CGI script, the Web server initiates a new process, which is terminated at the end of the execution of the script. Process creation and termination are very costly activities. Additionally, terminating the process after each request makes it difficult to retain information about the user session, unless such information is stored in a database, which again impacts performance. Terminating the process also prevents the management of shared resources used by multiple users across multiple HTTP requests.

Such limitations were bypassed by extending the capabilities of the Web server with an application execution engine, that is a server-side execution environment where the programs in charge of building the HTTP response can be processed without being terminated after each request, and shared resources can be associated with one or more applications and concurrently accessed by multiple users. Such an extended architecture typically also offers a main memory environment for storing session data whose duration goes across multiple HTTP requests.

### 3.7.1 Servlets

An example of extended Web server architecture is [JavaSoft's Servlet API](https://example.com), which associates the Web server with a Java Virtual Machine (JVM). The JVM supports the execution of a special Java program, the **servlet container**, which in turn is in charge of managing session data and executing **Java servlets**.

**Fig. 3.6** Java servlet architecture.
Java Server Pages (JSP) is a simple but powerful technology that extends Java servlets. A JSP page is composed of blocks of static code (HTML, JavaScript, CSS, etc.), mixed with dynamic blocks, i.e., portions of Java code executed by the scripting engine.

Fig. 3.7 The translation of JSP pages into servlets.
It stems from the early days of the Web, when it was used for CGI scripting. Now, it is a dynamically typed multi-paradigm programming language, freely available on most platforms. Like PHP, it is open source, and enjoys the support of a large community.

Java Server Pages (JSP) is a simple but powerful technology that extends Java servlets. A JSP page is composed of blocks of static code (HTML, JavaScript, CSS, etc.), mixed with dynamic blocks, i.e., portions of Java code executed by the scripting engine.

Each time the Web server receives a request for a same JSP page, it verifies whether changes to its code occurred since its creation by checking the content of the `Last-Modified` HTTP header. If not, the servlet instance for that page stored in memory is recalled; otherwise, the JSP page is recompiled and a new servlet instance is created and stored in memory. For this reason, the first access to a JSP page requires a longer time (the so-called first-person penalty), while the following requests are faster.

**Fig. 3.8** The structure of JSP pages.
With the advent of the Web, more and more data are available on-line, and Web applications no longer source the data they use to provide their services from a single, proprietary data source only. Increasingly, applications integrate data sourced from the Web, e.g., to translate IP addresses into logical location information (e.g., a country and a city) or similar. As we will see, sourcing and integrating data from the Web is one of the key features of mashups and, depending on how data are published, the complexity of this task may vary from relatively simple to very complex. For instance, sometimes data are available only via the HTML markup of a web application, and reusing them requires extracting them from the markup, a task that is non-trivial in general.

Three-tiered architectures need an execution environment supporting interlayer communication. Application servers provide such an environment.

By providing an intermediate layer between the Web server and the back-end for resource management, they enable the efficient execution of components in the application logic layer, thus supporting the construction of dynamic pages according to the execution flow depicted in Figure 3.9.

---

**Fig. 3.9 Application server architecture**
If a user issues a page request to the Web application, the request is intercepted by the Controller, which is in charge of deciding which business operation needs to be performed. The Controller, hence, invokes the respective component in the Model, which contains the necessary logic to execute the requested action, to update the state of the application, and to assemble the data to be presented to the user. The change of the application state activates the View, which fills the presentation template with the data computed by the Model. Finally, the so-constructed HTML response is sent to the client browser.

There are several application frameworks offering support for implementing the MVC pattern. Apache Struts is one of these frameworks. It is open-source and is based on Java-EE application server architecture. In Apache Struts, the Model represents the underlying data, the View is the actual rendered page, and the Controller handles HTTP requests, selects and invokes application logic, and performs page selection (i.e., handles navigation logic). For the Controller, Apache Struts relies on servlet technology and provides a default servlet implementation for this purpose, while it suggests the use of JavaServer Pages for a template-based approach to the View construction (although other technology may be used). Apache Struts also provides its own tag library so fostering additional expressibility.

It is worth noting that mashup development naturally follows the MVC design pattern. The Model is the application domain supplied by the Web-delivered services on which the mashup components are based, the View is the layer in charge of displaying the user interfaces by rendering presentation code, like HTML or JavaScripts, and the Controller is the middle interaction logic between Model and View, which operates on the model, for example requesting some new data from the involved components, and propagates the changes to the view, and vice versa. In mashups, such a pattern can be achieved both at the client side and at the server side. In fact, MVC is a design pattern, and as such it is not necessarily based on a multi-tier architectural pattern as the one described in the previous section. In a client-side mashup, for example, the logic for composing a map service with an additional data source, including the invocation of the two services, can be...
This chapter illustrated the most relevant technologies, standards, and specifications for the Web, with the aim of presenting some important ingredients for the development of mashups delivered on the Web. We started from the basic protocols and languages, such as HTTP and HTML. Then, we discussed how combinations of technologies (HTML, DOM, JavaScript, AJAX), and client-side business logic. We then extended the capabilities of HTML as a document markup language, allowing for sophisticated interactive features and client-side business logic. We then illustrated some important server-side technologies for building pages dynamically (such as servlets, server-side scripting, and tag libraries).

### 3.10 Summary and Bibliographic Notes

Many resources available online today, indeed, provide JSON-encoded data. Adoption of JSON to consume data coming from remote services and APIs is accommodated with JavaScript, including Web mashups, can take advantage of the JSON data to produce native JavaScript objects. A parser, a JavaScript program can use the built-in eval() function to interpret the structure of a file menu.

![JSON](http://www.json.org/example)

**Fig. 3.11** Examples of JSON and XML specifications. The represented data refer the structure of a file menu (http://www.json.org/example)