Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmermann**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

The Web as a Software Connector

Integration Resting on Linked Resources

Cesare Pautasso and Olaf Zimmermann

DECOMPOSING A SOFTWARE system into a set of distributed services to be integrated with each other has been and still is a wicked problem, despite the field's presumed maturity.[1] Thousands of APIs use the web today. However, HTTP's expressiveness and universality make it not only possible but also easy to use HTTP to implement many integration styles.

Here, we discuss how the web, seen as a graph of linked resources shared between microservices, can serve as a complementary integration style. Carrying this insight in your toolbox will make you aware of all the options to consider next time you build loosely coupled integrated systems.

The State of the Practice

File transfer, shared databases, remote procedure calls, and asynchronous messaging are the four classic integration styles commonly used to stitch together heterogeneous, autonomous, and distributed software systems within and across enterprises (see Figure 1).[2] Together with data streaming, they support different forms of dataflow and control flow across multiple integrated systems. They also impact the degree of coupling introduced in the integrated architecture.

Over the past decades, we've witnessed increased adoption of HTTP[3] as a convenient, ubiquitous network protocol for web-scale integration. First, HTTP was used (some might say misused) as a firewall-tunneling solution for implementing remote procedure calls and messaging across the web. HTTP is expressive; it can be and has been used to implement file transfers, streaming, and shared databases as well. RESTful HTTP[4] became popular in the mid 2000s. (REST stands for Representational State Transfer.) It uses the original features of the HTTP standard such as URIs, methods (`GET`, `PUT`, `POST`, and so on), hypermedia, and content type negotiation to address recipients, control behavior, and represent information exchanged.

Still, some confusion remains on how to properly use the web to build loosely coupled integrated systems. The rest of this article aims to clear up that confusion.

For the most part, hypermedia drives the design of RESTful APIs that help individual clients dynamically discover the business protocol for interacting with them. Nevertheless, we noticed that many integration design discussions online and in project rooms still focus on single interactions between the individual client and a single API. These happen using synchronous HTTP request–responses—in other words, hypermedia-driven remote procedure calls.

However, when you look at the integration's end-to-end result, in which multiple clients share and manage their common state through one or more linked web resources, a different picture emerges. You need this broader focus if you're to successfully use the web as an advanced integration style—for instance, in monolith decomposition and microservice recomposition, enterprise application integration, or business-to-business conversations. Choosing such a style is appealing when data should be not only transferred between parties (as with file transfers or message queues) but also published so that it can be retrieved, updated, or deleted by any number of parties (as with a globally distributed shared database).

Dataflow

To view the web as an integration style, we need to go back to the basics of what it means to exchange messages in integrated systems. Messaging aims to deliver

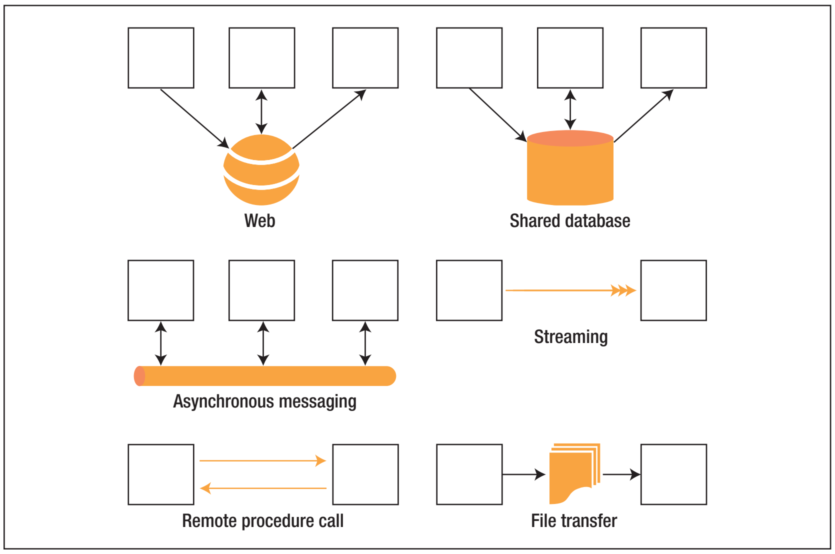

FIGURE 1. Software integration styles. The boxes represent services or microservices being integrated with each other. The arrows indicate the data flowing between them, sometimes directly (as with data streaming or remote procedure calls) and sometimes indirectly (as with the web, asynchronous messaging, file transfer, or shared databases).

data from the sender to the recipient (see Figure 2a) with certain guarantees. Asynchronous-messaging middleware (also known as distributed message brokers implementing the Message Bus pattern[2]) conveniently deals with all kinds of problems along the way. For example, it can deal with unavailable recipients (by store-and-forward techniques such as queueing), network failures (by retransmission and deduplication), and security attacks (by ensuring messages can't be forged or tampered with).

Once the message safely arrives at an endpoint, its recipient consumes it and the message, as a first approximation, disappears from the messaging system. This might happen as part of a transaction. (In advanced scenarios, recipients might only peek at a message without removing it from the queue. With event sourcing, the message queues might even turn into distributed event logs collecting all the messages that have ever been exchanged.)

Messages can be delivered to one or more recipients, and recipients can be addressed by a variety of schemes (for example, message queue names, subscription topics, or recipient lists). So, in some cases, recipients and senders don't need to be aware of one another. This is a facet of loose coupling.

Queue-based messaging is asynchronous, which further reduces coupling because senders and recipients don't have to be up and running at the same time for the message exchange to be successful. This is because the messaging middleware acts as an intermediary.

The web, on the other hand, was born as a global document-sharing platform. Web resources, identified by global addresses, hold information that can be published, read, and updated multiple times by multiple interested parties and deleted once it's no longer relevant. When viewed as an integration style, the web helps construct a global, shared data structure (a blackboard[5]) between the systems that need to be integrated (see Figure 2b). One system publishes information on the web so that others can retrieve it later. Another system might update the shared information, and yet another one might remove it once the previous system has processed it (see the bottom right of Figure 2b).

For example, Doodle lets you create a poll resource by publishing the available options when multiple parties need to agree on something (for example, a meeting location and time).[6] The link to the poll is disseminated among the interested parties, who can retrieve the options and express their preferences. Once all the votes are collected, the poll is updated to reflect the decision outcome. As long as all parties share the link to the poll resource, they can proceed to read or write from it in a completely asynchronous and uncoordinated manner.

The web wasn't designed to deliver information from senders to recipients with the same quality guarantees as queue-based messaging. Rather, it was designed to publish information from one sender to be read by potentially millions of recipients either forever or until the information is removed from the web.[7]

Information stored in web resources can be accessed with multiple representations, fostering format interoperability (because every system can use content type negotiation to retrieve the most appropriate and compatible representation). In messaging, format interoperability is achieved by either standardizing the message format or performing message translation. (Network protocol interoperability is a different concern that's addressed with the corresponding HTTP or Advanced Message Queueing Protocol [AMQP][8] standard.)

Seen as a global shared data structure meant to be concurrently accessed by multiple systems, the web provides only optimistic locking because it assumes that most interactions are read-only. So, the level-of-consistency guarantees it provides aren't comparable to the ACID (atomicity, consistency, isolation, and durability) model that is, for instance, prominently promoted in the shared-database integration style.[2,9] Still, you could view the web as a globally distributed key–value store in which every key (URI) can potentially be mapped to a separate webserver. To deal with nonuniform access patterns such as key hot spots, you can introduce layers of caching to mirror popular keys, again assuming that the corresponding values don't change often.

Such a massively scalable system deals with discovery through *decentralized referral*, in which the keys of related resources are found thanks to hypermedia. Links are embedded in a representation of a resource pointing to one or more related resources, without any guarantee that the related resource actually exists. Messages in asynchronous-messaging systems also carry unique identifiers and can point to related messages (for instance, through Correlation Identifiers[2]). However, those identifiers server other purposes related to message delivery guarantees (primarily to avoid duplicate delivery and establish chains of replies or conversations).

In terms of the time dimension of coupling, the web shares the asynchronous nature of messaging. Systems publishing information on a particular web resource can be long gone when other systems retrieve that information from the shared web resource. In other words, relaying information through a shared web resource requires two synchronous HTTP request–response interactions with the resource (see Figure 3) but no direct interaction between the sender and receiver. Such behavior also occurs between two systems integrated through a message queue.

Likewise, as long as the systems agree on the shared resource's address, they don't need to be aware of one another because they use the resource to exchange information. In messaging, the queue address must be correctly configured for both the sender and recipient. On the web, thanks to hypermedia, resource

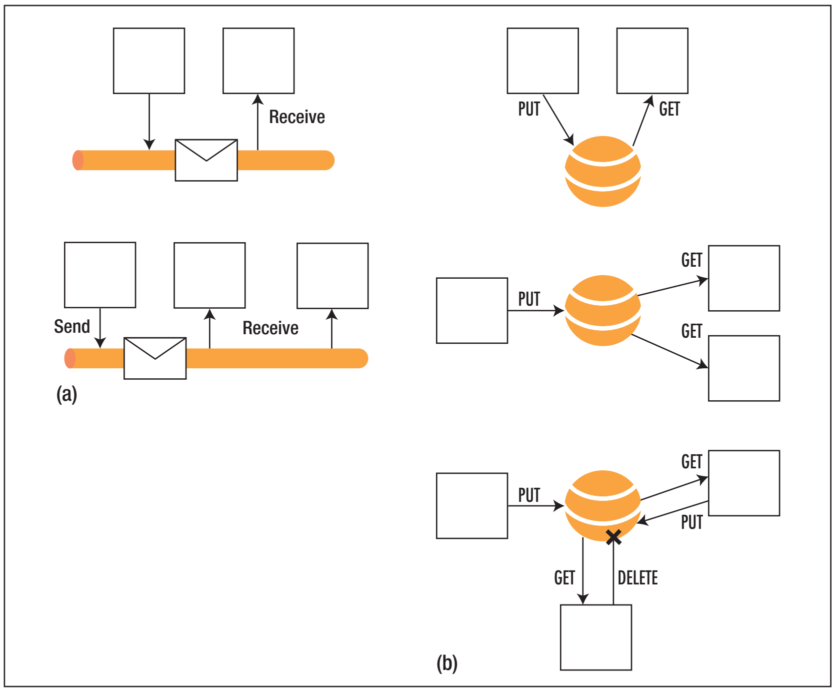

FIGURE 2. Dataflow integration primitives. (a) Asynchronous messaging. The top shows a message delivered to one recipient; the bottom shows a message delivered to multiple recipients. (b) The web. The top shows a resource shared between one writer and one reader, the middle shows a resource shared by one writer and two readers, and the bottom shows a complex multiparty conversation (with **GET**, **PUT**, and **DELETE**) over the same shared resource.

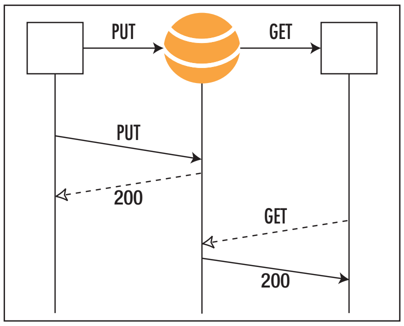

FIGURE 3. Refining the conceptual end-to-end dataflow of the web integration style (from Figure 2) to the HTTP network protocol, as a sequence of basic request–response interactions. (200 is the HTTP OK status code.)

identifiers (URIs) can be dynamically discovered by being looked up from other web resources (playing the role of directory). URIs have global validity and, thanks to the Domain Name System (DNS), are assigned in a decentralized manner. Message queue identifiers are valid within the context of one particular messaging system, where naming conventions such as `topic://unique.queue.name` can be used but aren't enforced globally.

Control Flow

When messages arrive at a recipient, they're processed. The message delivery event triggers the execution of the code in charge of consuming the message. In other words, sending a message not only transfers information from the sender to one or more recipients but also explicitly or implicitly provokes the invocation of processing logic in the recipient.

Unlike with remote procedure calls, the transfer of execution (or control) flow is asynchronous in messaging. Senders and recipients don't have to be available at the same time, and the sender doesn't have to wait for the message delivery to complete before carrying on. Recipients either can be notified as soon as messages arrive (push control flow) or will need to periodically check their inbox (pull control flow). Furthermore, the implementation details of the server-side processing logic are hidden from the sender and the messaging infrastructure.

On the web, similar strategies are possible to react to state transitions of shared resources. Resources can be polled so that code executes when interesting updates are detected (pull). Conversely, it's possible to intercept resource state changes and execute code as soon as they happen (push). This used to be possible only if the code was deployed "together" with the resource on the webserver hosting it. Additionally, WebHook callbacks let resources notify interested clients when state changes occur. Also in this case, the client and the HTTP server are unaware of the implementation platform that executes the processing logic.

The web of linked resources is a global blackboard for asynchronous application and service integration.

Comparison Criteria

Table 1 compares asynchronous messaging and the web regarding quality attributes and other decision-making criteria. The table is organized according to viewpoints and cross-cutting quality concerns.

Asynchronous messaging and the web share many characteristics in terms of

- the many-to-many integration topologies they enable,
- the data representation syntax's flexibility,
- the presence of standard network protocols (such as HTTP and AMQP), and
- the coupling implications (they're both asynchronous from an end-to-end perspective).

However, they differ fundamentally in terms of the basic abstraction (message versus resource). This difference impacts the semantics of the primitives provided to perform the integration (for example, send–receive versus `GET–PUT–DELETE`). Specifically, with messaging it's possible to notify recipients of a message's arrival (push control flow), whereas on the web, microservices interested about resource state transitions must resort to polling unless they're deployed on the same webserver hosting the resource. WebHooks are a common, albeit clumsy, way to work around this limitation.

Other differences concern how to achieve payload interoperability (message translation versus content type negotiation) and achieve reliability (message retransmission and duplicate removal versus retrying `GET`, `PUT`, and `DELETE` requests whose idempotence is made explicit in the protocol).

Overall, a conscious architectural decision selecting an integration style is necessary, providing rationale rooted in the project context and requirements, as Uwe Zdun and his colleagues described.[10]

Most research in applying HTTP for software integration focuses on understanding and controlling the interaction and state transfers between one client and the individual resources that make up RESTful

Table 1. A comparison of asynchronous messaging and the web regarding quality attributes and other decision-making criteria.*

Criterion	Aspect	Asynchronous messaging	The web
Logical viewpoint (defining concepts)	Integration abstractions	• Message • Message queue	• Resource • Graph of linked resources (hypermedia)
	Integration architecture	• Message endpoints as clients of a messaging server • Federation of message brokers	Client-server
Implementation viewpoint	Transfer syntax	Any (text, binary, and SOAP XML)	Any IETF RFC'ed media type (for example, JSON and XML, but also binary)
	Endpoint-to-broker API primitives	• Nine AMQP performatives (including `open`, `transfer`, and `close`) • JMS API calls (for instance, implementing message selectors) • Send and receive • Publish, subscribe, and notify	HTTP verbs (methods) such as `GET`, `PUT`, `DELETE`, and `POST`
Deployment viewpoint	Single-connector topology	1:*N* (many endpoints per messaging broker)	1:*N* (many clients to a shared resource)
	End-to-end topology (endpoint landscape and services)	*M*:*N* (Multiple senders can reach multiple recipients.)	*M*:*N* (Multiple microservices share common resources.)
Process viewpoint	Dataflow	Yes, via the message payload	Yes, via resource representations
	Pull control flow	Yes (queue polling)	Yes (resource polling)
	Push control flow	Yes (message delivery notifications)	Yes (only on the same webserver or using WebHooks)
Coupling dimensions and quality attributes (cross-cutting concerns)	Coupling in time (end to end)	Asynchronous (The message queue decouples the message's sender and recipients.)	Asynchronous (The resource decouples the state transfer's source and destinations; see Figure 3.)
	Addressing	Local to each messaging middleware (message queue names)	Global (resource URIs)
	Discovery	• Directory lookup • Referral via protocol headers	• Directory lookup • Referral via links in the payload
	Format interoperability	Message transformation patterns such as the Canonical Data Model and Message Translator[2]	Multiple resource representations (content type negotiation)
	Network protocol interoperability	• AMQP • Proprietary interbroker protocols	HTTP
	Reliability	• Store-and-forward • Message retransmission • Duplicate removal	• Request–retry • Idempotency
	Security	Message-level and/or in-or-by messaging middleware plus underlying protocols	• Message-level and/or HTTPS (TLS/SSL) • OAuth, OpenID, and so on
	Transactions	Endpoints may (but don't have to) participate in single-phase-commit and two-phase-commit transactions (for ACID-style strict consistency).	Various models have been proposed (such as Transaction as Resource, WebDAV, and Try-Cancel/Confirm).[9]

** IETF RFC = Internet Engineering Task Force Request for Comments, JSON = JavaScript Object Notation, AMQP = Advanced Message Queueing Protocol, JMS = Java Message Service, TLS/SSL = Transport Layer Security/Secure Sockets Layer, WebDAV = Web Distributed Authoring and Versioning, and ACID = atomicity, consistency, isolation, and durability.*

ABOUT THE AUTHORS

CESARE PAUTASSO is a full professor at the new Software Institute at the University of Lugano's Faculty of Informatics. Contact him at c.pautasso@ieee.org.

OLAF ZIMMERMANN is a professor and institute partner at the University of Applied Sciences of Eastern Switzerland, Rapperswil. Contact him at ozimmerm@hsr.ch.

HTTP hypermedia APIs. In this article, we reminded you to look at the bigger picture of end-to-end integrations of multiple parties such as service and microservice compositions, which asynchronously maintain a shared data structure published on the web. This usage and reinterpretation makes the web a first-class citizen in the land of integration styles. This complementary style shares many of its advantages with asynchronous, queue-based messaging. But, as a shared-memory blackboard, it opens additional opportunities owing to the massively scalable and decentralized nature of the web.

There's more to integration than choosing an integration style. Once you decide how to control the coupling by introducing an appropriate integration style, more decisions are necessary. For example, how do you decompose the monolith? (One way is to look for bounded contexts with strategic domain-driven design.[1]) How do you define the corresponding business protocols? (One way is with conversations and interaction sequencing rules.) How do you design the individual interactions? (One way is to use interface representation patterns.)

References

1. C. Pautasso et al., "Microservices in Practice, Part 1: Reality Check and Service Design," *IEEE Software*, vol. 34, no. 1, 2017, pp. 91–98; ieeexplore.ieee.org/document/7819415.
2. O. Zimmerman et al., "A Decade of *Enterprise Integration Patterns*: A Conversation with the Authors," *IEEE Software*, vol. 33, no. 1, 2016, pp. 13–19; ieeexplore.ieee.org/document/7368007.
3. "Hypertext Transfer Protocol—HTTP/1.1," Internet Soc., 1999; tools.ietf.org/html/rfc2616.
4. R.T. Fielding et al., "Reflections on the REST Architectural Style and 'Principled Design of the Modern Web Architecture,'" *Proc. 11th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 17), 2017, pp. 4–14; research.google.com/pubs/archive/46310.pdf.
5. F. Buschmann et al., *Pattern-Oriented Software Architecture*, vol. 1, John Wiley & Sons, 1998.
6. "What Is Doodle and How Does It Work: An Introduction," Doodle, 2017; help.doodle.com/customer/portal/articles/761313-what-is-doodle-and-how-does-it-work-an-introduction.
7. T. Berners-Lee and M. Fischetti, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*, Harper Business, 2000.
8. "ISO and IEC Approve OASIS AMQP Advanced Message Queuing Protocol," OASIS, 1 May 2014; www.oasis-open.org/news/pr/iso-and-iec-approve-oasis-amqp-advanced-message-queuing-protocol.
9. N. Mihindukulasooriya et al., "A Survey of RESTful Transaction Models: One Model Does Not Fit All," *J. Web Eng.*, vol. 15, no. 1, 2016, pp. 140–169.
10. U. Zdun et al., "Sustainable Architectural Design Decisions," *IEEE Software*, vol. 30, no. 6, 2013, pp. 46–53; ieeexplore.ieee.org/document/6576117.