Composing RESTful Services and Collaborative Workflows: A Lightweight Approach

Article *in* IEEE Internet Computing · October 2008

DOI: 10.1109/MIC.2008.98 · Source: IEEE Xplore

CITATIONS
88

READS
106

4 authors:

Florian Rosenberg
Crayon AI Center of Excellence
66 PUBLICATIONS **3,993** CITATIONS
SEE PROFILE

Francisco Curbera
IBM
107 PUBLICATIONS **11,780** CITATIONS
SEE PROFILE

Matthew J. Duftler
IBM
18 PUBLICATIONS **1,678** CITATIONS
SEE PROFILE

Rania Khalaf
IBM Research
77 PUBLICATIONS **4,112** CITATIONS
SEE PROFILE

Composing RESTful Services and Collaborative Workflows

A Lightweight Approach

The use of RESTful Web services has gained momentum in the development of distributed applications based on traditional Web standards such as HTTP. In particular, these services can integrate easily into various applications, such as mashups. Composing RESTful services into Web-scale workflows requires a lightweight composition language that's capable of describing both the control and data flow that constitute a workflow. The authors address these issues with Bite, a lightweight and extensible composition language that enables the creation of Web-scale workflows and uses RESTful services as its main composable entities.

Florian Rosenberg
Technical University Vienna

Francisco Curbera, Matthew J. Duftler, and Rania Khalaf
IBM T.J. Watson Research Center

The increasing adoption of the service-oriented architecture (SOA) paradigm enables service composition and scalable Web processes (or workflows) as a means of building distributed applications. These systems use standardized technologies and protocols, such as the Web services stack (including SOAP, the Web Services Description Language [WSDL], and Web Services Business Process Execution Language [WS-BPEL] specifications[1]).

The academic and industrial community is still engaged in a lively, ongoing debate about the limits of the Web services model, focusing on two main contention points: the Web services stack's perceived complexity and its lack of alignment with the Web's accepted architectural principles (described elsewhere[2,3]). Much of the complexity attributed to Web services is in fact tied to requirements from traditional enterprise computing, in particular those associated with quality of service (QoS) management.[3] This explains the natural receptivity to the Web services stack in enterprise settings and the corresponding lack of enthusiasm among pure Web developers.

Despite numerous transgressions in daily Web practice, the Web development community consensus widely backs the resource-oriented paradigm as proposed by the Representational State Transfer (REST) model, which was introduced as an architectural style for distributed hypermedia systems.[4] In REST, the main architectural concept is the information resource, and the prin-

1089-7801/08/$25.00

cipal interaction mode lets clients retrieve representations of those resources. REST imposes some constraints, such as a *uniform interface*, meaning that all resources present the same interface to clients, and *protocol statelessness*, meaning that the server keeps no state on the client's behalf. As Steve Vinoski points out in an overview of REST vs. WS-*, "The fact that the Web works as well as it does is proof of these constraints' effectiveness."[5]

The more important aspect of this discussion probably lies in how each architectural paradigm (and each developer community) can benefit from the other. The simplicity of the single interface, single protocol (HTTP in REST) approach has potential benefits in enterprise scenarios, if architects and developers can overcome QoS concerns. In this article, we address the complementary problem: how the service paradigm and SOA's service composition model can benefit mainstream Web applications and their development model. In particular, we explore the potential benefits of process-oriented composition for Web scenarios. We focus on the design principles and applications of a lightweight, process-oriented composition model — the Bite language — that we derived by aligning SOA process composition principles with REST architectural requirements.

Bite Overview

Architecturally, the Bite model integrates first-class awareness of REST principles into a simplified workflow language. Its power becomes evident, however, when combining SOA's integration capabilities with Web-centric applications. Bite lets Web developers create applications that can seamlessly combine

- *RESTful services*, such as Atom feeds and Web queries. Graphical flow models already available on the Web, such as Yahoo Pipes (http://pipes.yahoo.com/pipes/), enable a dataflow composition model for feeds but are unable to incorporate other interaction types.
- Simple *human interaction*, as supported by forms, instant messaging, and linked email exchanges. Most Web applications are naïvely interactive and collaborative.
- *Collaboration services*, such as Lotus Activities (www-306.ibm.com/software/lotus/products/connections/activities.html)orLotus Quickr (www-306.ibm.com/software/lotus/products/quickr/), which support rich, unstructured interactions between ad hoc communities linked via common business goals. Beyond individual interactive primitives, complex collaborative applications are a key characteristic in social Web applications usually included under the Web 2.0 banner. Integrating structured processing with unstructured collaboration showcases process-centric composition's value in the Web environment.
- *Back-end services*. Process composition constitutes the backbone for SOA-based process automation in the enterprise by enabling a model for back-end service integration. In a Web-centric environment, process composition brings structured access to back-end services into traditional interactive and collaborative applications with a simple, integrated programming model.

Here, we'll focus on Bite's design principles, its extensibility, an application scenario for collaborative flows, and the Bite runtimes that support multiple platforms.

The Bite Approach

Bite aims to provide a lightweight and extensible language and integrated programming model for implementing RESTful service composition and interactive flows. We introduce Bite's basic motivation and detail its core constructs elsewhere.[1] In this article, we build on that work.

Requirements and Design Goals

Bite is a process composition model for Web applications, so its foremost requirement is the alignment with REST architectural principles (see our previous work for further details[1]). Beyond that, Bite focuses on enabling an agile, iterative, and community-oriented development approach by adhering to the following design goals and requirements:

- *Atom life-cycle model*. Deployed Bite processes are collections whose members are all running process instances, created by HTTP `POST` requests sent to the collection URL. Management operations retrieve either the process instance list (`GET` against collection) or an individual process instance execution state (`GET` against collection member). Processes are thus resources in their own

right, but they also interact with other resources through first-class HTTP activities (GET, POST, PUT, DELETE). Likewise, each deployed process in a Bite runtime belongs to the collection of Bite-deployed processes; Bite uses POST requests to deploy new process models.

- *Lightweight process model*. Existing composition approaches are still very complex, so Bite requires a lightweight model and execution engine that's still powerful enough to implement real-world Web-scale workflows (called *flows* in Bite). The Bite language has only a basic set of predefined language constructs for specifying the flow logic but doesn't support scopes, compensation, or transactional behavior. However, it still provides a rich execution semantics that is aligned with the semantics of WS-BPEL activities that can occur in a WS-BPEL flow activity.
- *Scripting approach*. Bite adopts many concepts from traditional scripting languages, such as dynamic data types. In Bite, users don't need to explicitly define or type variables before they can use them. Incoming or outgoing activity data is implicitly available (once a user has defined a data link) and is dynamically typed. Bite uses convention over configuration, such as having default I/O variables for each activity.
- *Language extensibility*. One key feature of Bite is its extensibility, which lets the community add new activity types by implementing the required code in a programming or scripting language. Following the scripting approach, Bite enables writing such extensions in any major scripting language (such as Groovy, Python, or Ruby). This lets users create a large catalog of extensions.
- *Web and human integration*. Another important aspect is that Bite integrates humans that participate in flows. This requires Web front ends that can interact with the flow to allow a so-called deep integration with the Web.

By addressing all these requirements, Bite can be used in a flexible manner to address various kinds of workflow applications.

Basic Model and Language

The basic Bite process model comprises a flat graph (except for loops) containing atomic actions (activities) and links between them. To create loops, we use a dedicated `while` activity, the only construct allowed to contain other activities. Bite encodes graph execution logic in conditional transition links between activities and supports error handling via special error links to an error-handling activity.

A key Bite feature is its small set of basic activities. The language's core activities consist of

- basic HTTP communication primitives for receiving and replying to HTTP requests (`receiveGET/POST`, `replyGET/POST`) and making requests to external services (`GET`, `POST`, `PUT`, `DELETE`);
- utility activities for waiting, calling local code, or terminating the flow; and
- control helpers, such as external choice and loops.

A Bite flow both uses external services in its flow logic and exposes itself as a service. Sending an HTTP `POST` request to a flow's base URL creates a new flow instance that's assigned a new instance URL. This instance URL is returned in the response's HTTP `Location` header field. The instance URL contains a flow ID to correlate subsequent requests to that flow.

Each flow instance can define multiple `receive` activities corresponding to multiple entry points. These activities expose additional URLs as logical addresses of the instance's nested resources. The Bite runtime dispatches `POST` requests directed to these URLs to the individual `receive` activities in the flow model using the relative URLs defined in the activities' `url` attribute. This mechanism allows users to build interactive flows that expose multiple entry points for interacting with the flow — for example, by using different Web forms that act as a front end for the different flow entry points. To support the development of dynamic Web forms that interact with the flow (to access the state and its variables), we provide a set of tag libraries for developing dynamic Web pages using Java Server Pages (JSPs).

Flow example. Listing 1 depicts an example that aggregates two RSS news feeds and also shows the use of Bite extension activities. Note that XML is only one possible representation of a Bite flow; we also support a Groovy DSL variant. Each flow is specified within a `<process>` element containing an optional `expressionLanguage` attribute.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <process name="feeds">
 3   <receivePOST name="rssRcv" url="feeds"/>
 4   <GET name="getYahooFeed"
 5              target="http://rss.news.yahoo.com/topstories">
 6   </GET>
 7   <GET name="getBBCFeed"
 8              target="http://newsrss.bbc.co.uk/rss.xml">
 9   </GET>
10   <aggregate name="aggregateFeeds">
11     <input value="$:getBBCFeed"/>
12     <input value="$:getYahooFeed"/>
13   </aggregate>
14   <replyGET name="rssReply" url="feeds">
15     <input value="$:aggregateFeeds"/>
16   </replyGET>
17   <sendMail name="notify" address="user@example.com">
18     <input value="aggregation has taken place"/>
19     <control source="aggregate"/>
20   </sendMail>
21 </process>
```

Listing 1. Feed aggregator flow. This shows how to aggregate two news feeds and send an email upon completion.

This flow is triggered when a user or an application sends a `GET` request to the relative URL specified in the `url` attribute of the `receiveGET` activity — for example, http://localhost:8080/bite/feeds for a flow deployed on the `localhost` in the bite directory.

After the instantiating request, the Bite runtime activates the next two REST activities (lines 4 through 6 and 7 through 9) because they don't depend on any activities. Each issues a `GET` to an external resource — the Yahoo and BBC news feeds (URLs shortened for space reasons) — and places the resulting RSS feed document into an implicit variable, `getYahooFeed_Output` and `getBBCFeed_Output`, respectively.

The next activity, called `aggregate`, is an extension activity that takes two RSS XML documents as input and aggregates both feeds into one. The `aggregate` activity defines two input elements referring to the two preceding activities, thus defining both a data and an implicit control link (because the name refers to an activity). If an attribute value is an expression, not a literal, then it's prefixed with a `$:` to indicate the Bite runtime that it must have to be evaluated. We can locally override the process element's expression language by specifying the desired expression language directly — for example, using `$groovy:` to use Groovy for evaluating this expression.

After aggregating the feeds, the flow returns the result to the caller and sends an email to the administrator. (Notice the explicit control link between `sendMail` and `aggregate`.) Then, the flow instance is finished.

Bite extensibility. Bite's extensible design lets the developer community provide additional functionality in a first-class manner by creating Bite *extension activities* and registering them with the Bite engine. This design lets us keep the language and its runtime very small, and lets developers implement other required activities as extensions. Bite facilitates the implementation of extension activities using Java or any scripting language that the Java Scripting API supports.

Creating an extension activity lets developers implement the core extension logic with minimal overhead for reading and evaluating the extension's syntax. Bite passes all activity attributes and inputs to the implementation in a map object, with an ordered list for unnamed inputs. For example, the map for the `aggregate` activity in Listing 1 has the name attribute's value. The extension activity implementation uses the values of inputs and attributes, does its work, and returns an instance of an invocation result object containing the result (an XML object, in the `aggregate` case). To register the implementation with the Bite runtime, the creator simply deploys it into a predefined directory for all extensions or keeps it in the same location as the flow. The extension's name must be unique and is defined by its filename.

Executing scripts. Executing script code in the flow facilitates certain tasks (for example, calculation or accessing some back-end system). So, we have a `script` extension that lets Bite execute scripts written in any scripting language that Bite supports, defined either inline or in an external file. The scripts are fully integrated in

```
<script file="renderFtl.groovy" />
...
<replyGET name="htmlReply" url="reports">
   <input value="$:renderFtl('reports.ftl')"/>
</replyGET>
```

Listing 2. Rendering sample. This sample shows how to include rendering capabilities in Splice using a script.

the flow and thus have access to activity variables and can define links to other activities.

UI integration and rendering capabilities. Bite enables a very flexible user interface (UI) integration because all entry points in a flow are accessible via URLs. Integrating with a mashup tool is thus quite straightforward: parts of the mashup that need to interact with the flow make an HTTP call to the appropriate URL or provide such a capability via HTML links or buttons for the user to select. The mashup tool might provide a dashboard containing several widgets, each of which might interact with the flow itself. For example, a Web application that interacts with several stores might be exposed using a widget showing a Google map of different stores, another aggregating available store items, and a third letting users reserve an item for pick-up. As the user performs map-related functions (zooming, getting driving directions, and so on), the flow isn't involved. However, clicking to reserve the item sends an HTTP request to the flow itself. A mashup application (QEDWiki) that uses a Bite workflow in the back end is described elsewhere.[6]

In addition to the aforementioned UI integration, Bite provides flexible HTML rendering capabilities using various template languages to render HTTP replies or emails from a flow. Bite provides rendering capabilities using rendering functions that a developer must write once for a specific template language and that can then be imported if needed (using the aforementioned `script` activity). Listing 2 shows a simple example of using rendering capabilities from the FreeMarker template language (http://freemarker.sourceforge.net). Line 1 makes the script that provides the FreeMarker capabilities globally available and defines a function called `renderFtl`, which takes a template file as an argument. In the `replyGET` activity at the end of a flow, we can use the `renderFtl` function to render the response. The template itself has access to all the activity data it needs to render the data.

These rendering capabilities clearly separate the flow logic from the presentation logic, which have to be specified in separate template files. Additionally, the `script` activity lets users plug in any template language by implementing a specific rendering function (roughly 10 lines of code in the FreeMarker case).

Collaborative Flows in Bite

Besides writing data-driven applications that don't involve the user in the flow execution (as Listing 1 demonstrates), Bite provides built-in, first-class support for interactive, collaborative flows, specifically for email and browser-based interactions (forms or links). The most common interactions between users and Web applications aren't dependent on proprietary tools — they usually occur over a browser, via Web forms or an email client, for instance. A user fills out a Web form, for example, and after submitting the form, gets an email with further instructions that also contain a link. Clicking on that link provides another entry point to a specific state of the Web application (an order approval the department manager has conducted, for example).

Bite supports these concepts by providing entry and response points in the flow logic that are clearly specified using `receive/reply` activities with relative URLs. The `sendMail` activity allows sending emails directly from the flow. A reserved variable containing the process instance URL easily lets users add links to email bodies or Web pages that can point back at subsequent entry points in the flow. Finally, the ability to create and plug in extension activities lets us add a library of activities that provide first-class support for any specific collaborative tools as needed. So, a single Bite flow and the corresponding HTML pages are often all that's needed to create interactive Web applications, such as ones that can place an order, get a manager's email from a RESTful service, request and get approval, and finally place the order if approved.

From Workflows to Collaborative Flows

Traditionally, a workflow provides structured behavior, but humans heavily use Web applications and, unlike machines, often accomplish a task in an unstructured manner. In such free-flowing tasks, actions and events could occur in multiple ways, multiple people could be involved, and several different media could exist that don't go

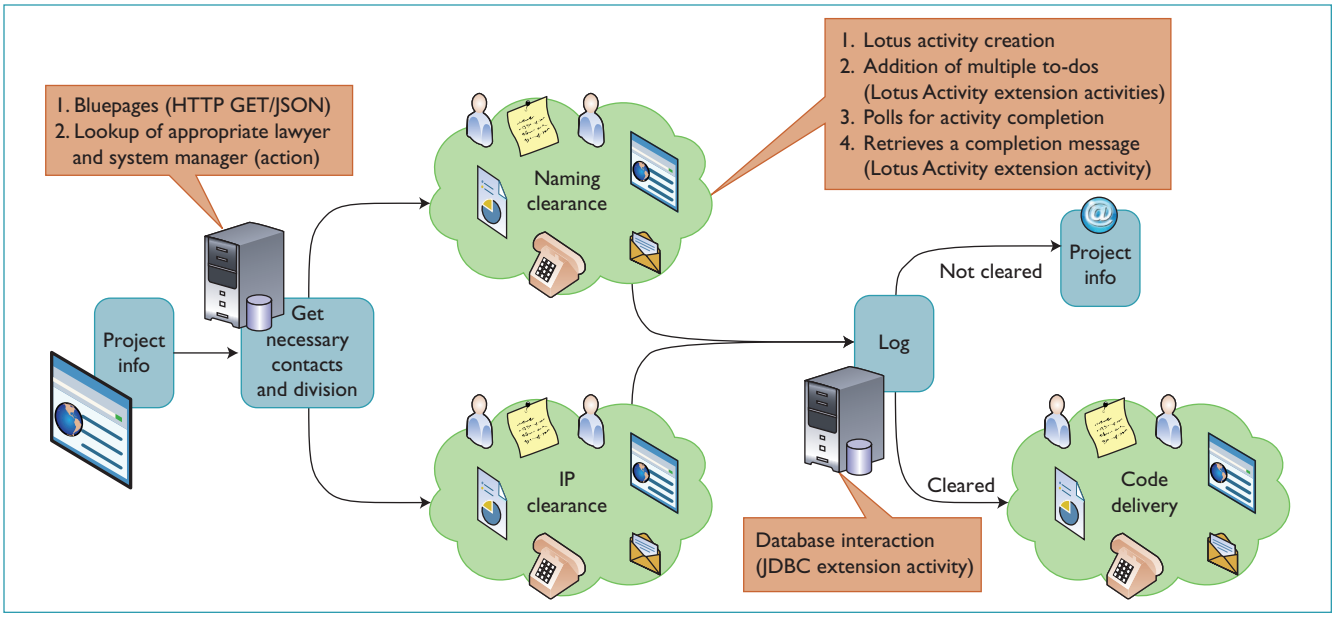

Figure 1. A collaborative Bite flow. Bite composes a flow for releasing software to the public by combining unstructured collaboration activities, back-end services, and user interactions with Web forms.

to a centralized application, such as email, face-to-face conversations, and phone calls.

Consider releasing a piece of software to the public. This might involve getting approval from a lawyer for intellectual property clearance or for the software's proposed name, or preparing and delivering the software and its related documentation to a distribution site. Each step, on its own, is a free-flowing task. It wouldn't be prudent to restrict a user to using a workflow to define such a task. Collaborative tools, such as Lotus Activities, can do a much better job at creating to-do lists, keeping track of completion, and putting the necessary people in touch with each other.

Instead of building unstructured collaboration capabilities directly into a structured workflow language or forcing a user to use structured flows in which a mix of structured and unstructured behavior is needed, we can achieve first-class integration with collaborative tools, such as Lotus Activities. We do this similarly to how we integrate forms but also add extension activities that act as the glue between the chosen collaboration tool and the Bite workflow.

A Collaborative Flow Example

The left side of Figure 1 shows the basic collaborative workflow that a team must carry out to release a piece of software to the public. This involves several unstructured tasks that don't themselves need a structured workflow. However, the workflow is beneficial in that it provides logic and back-end support for this group of related tasks. Such structural logic includes

- sequencing or providing conditions on a set of tasks — the team can't deliver the code ("code delivery," in the figure) unless they have completed the IP and naming clearances ("naming/IP clearance") and the lawyers approve them ("cleared" condition);
- affecting one task due to events in another, such as cancelling the naming clearance task if the IP clearance request is denied; or
- interacting with back-end systems to, for example, get the lawyers' names from the company directory or log the result ("lookup" or "log" via `GET`/`POST` activities).

Let's look at the flow in detail. It starts by collecting project information from the project team in a Web form that's posted to the flow (`receivePost` activity). This triggers a contacts lookup in the company directory ("lookup" in the figure via a `POST` activity). After the lookup, the flow triggers the "naming clearance" and "IP clearance" tasks, both of which are handled via the Lotus Activities server, which enables unstructured communication among participants (in this case, lawyers, project managers, and so on). Both tasks finish when all the unstructured activities (such as to-dos) are complete. The lawyer to whom the task is assigned must specify the outcome of these clearance tasks by adding

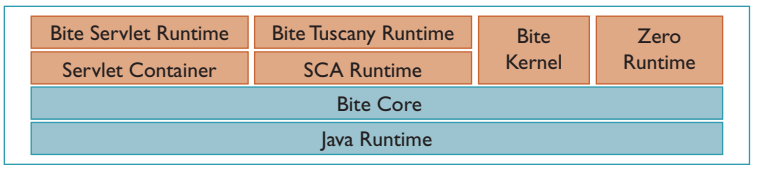

Figure 2. Bite runtime architecture. On top of the Bite Core, the runtime stack enables a range of hosting possibilities, such as a servlet container, a service component architecture-based runtime, or hosting on IBM's Project Zero.

a message "naming cleared" and "IP cleared" in the Lotus Activities system. Then, if the lawyer didn't grant the clearances, the flow sends an email to the requestor. Otherwise, the code delivery task begins and completes the same way the clearance tasks did.

This collaborative flow uses a set of extension activities to communicate with a collaborative platform — in our case, the Lotus Activities server. These include `createTask`, `createToDo`, `createMessage`, `waitForCompletion`, and others. These extension activities help users to create, edit, and complete collaboration tasks (in the figure, "naming," "IP clearance," and "code delivery"). Lotus Activities manages these tasks until a team member marks them completed, causing the flow to receive notification about the result.

This case study shows that Bite facilitates easily integrating collaborative tools directly by implementing the connectors as extension activities. These extension activities are simple; they contain mainly calls to the Lotus Activities server in the form of RESTful requests, with some pre- and postprocessing, such as authentication and extraction of information from the response messages.

Existing mashup solutions try to address some of these concerns as well, but they don't offer the level of flexibility and extensibility Bite does (see the "Related Work in Mashups" sidebar).

Runtime Architecture

We can execute a Bite flow on different runtimes depending on execution requirements. Figure 2 depicts the stack of supported Bite runtimes.

We implement the core part of Bite, the *Bite Core*, as part of a small layer on top of the Java Runtime environment. This part implements the execution and navigation logic for a Bite flow, the request handling, the expression framework, and the extension logic (among other things).

On top of the core logic, we provide four different runtimes that enable using Bite in scenarios with different requirements.

The *Bite Servlet Runtime* is suitable for hosting Bite flows as stand-alone Web applications. It's based on a modern servlet container (Apache Tomcat, in our environment) that implements a simple servlet, listens to HTTP requests and delegates the processing to the Bite Core, and returns the result to the servlet as an HTTP response.

The *Bite Kernel* provides an even simpler runtime directly on top of the Java platform and eliminates the need to deploy a Bite flow to a servlet container. The kernel allows an iterative development and testing of flows as well as embedding the flow execution into other applications. The implementation is similar to the servlet container — only a very small adapter is necessary to provide an HTTP listener for delegating the requests to the Bite Core and processing its response.

Zero Runtime provides the foundation for developing flows within Project Zero (www.projectzero.com), an IBM Community Source Initiative for building Web 2.0 applications based on REST principles. The Bite language is part of the Zero Assemble module and thus allows both the Bite runtime as well as the flow creators to leverage some of Zero's core features.

The *Bite Tuscany Runtime* allows the execution of Bite flows as part of a composite application implemented and enacted by using the *service component architecture*.[7] More specifically, we've used Apache Tuscany Runtime, but a detailed description is out of this article's scope.

Process composition in SOA has turned out to be the key technology for SOA-based automation and integration of business operations and scientific projects. Simultaneously, Web-based applications' variety and reach is bringing in new user demands and developer requirements. In particular, many Web applications need to integrate RESTful services and traditional back-end systems, as well as support rich user interfaces (incorporating forms, email, and so on) and emerging social collaboration tools. Incorporating the lessons learned from the SOA composition experience will endow the Web platform with a powerful and highly usable integration paradigm that can yield better developer productivity and more efficient and maintainable Web applications.

We've explored applying process-oriented composition to Web application development,

focusing on the Bite process model. Other composition models and SOA integration principles might also yield valuable lessons for the REST development community, just as the REST architectural principles and the Web platform as a whole are likely to fundamentally transform the enterprise computing world in years to come.

Acknowledgments

We thank Marc-Thomas Schmidt and Thomas Moran for their input on combining workflows with collaborative tools.

References

1. F. Curbera et al., "Bite: Workflow Composition for the Web," *Proc. 5th Int'l Conf. Service-Oriented Computing* (ICSOC 07), Springer-Verlag, 2007, pp. 94–104.
2. L. Richardson and S. Ruby, *RESTful Web Services*, O'Reilly, 2007.
3. S. Weerawarana et al., *Web Services Platform Architecture*, Prentice Hall, 2005.
4. R.T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, Univ. of California, Irvine, 2000.
5. S. Vinoski, "REST Eye for the SOA Guy," *IEEE Internet Computing*, vol. 11, no. 1, 2007, pp. 82–84.
6. M.-T. Schmidt, "Assembling the Perfect Feed," Jan. 2008; www.projectzero.org/wiki/bin/view/Community/MarcThomasBlog/BlogEntry1.
7. "Service Component Architecture Specifications," Open Service Oriented Architecture (OSOA), Jan. 2008; www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

Florian Rosenberg is a PhD candidate in the Distributed System Group at the Technical University Vienna. He performed this work while working as a research co-op at IBM T.J. Watson Research Center. His research interests include software composition, service-oriented architectures, and software engineering in general. Contact him at florian@infosys.tuwien.ac.at.

Francisco Curbera is a research staff member and manager of the Component Systems Group at the IBM T.J. Watson Research Center. His research interests include the use of component-oriented software in distributed computing systems. He has a PhD in computer science from Columbia University. Contact him at curbera@us.ibm.com.

Matthew J. Duftler is a software engineer in the Component Systems Group at IBM T.J. Watson Research Center. His research interests include component-based software engineering, Web services, and workflows. Contact him at duftler@us,ibm.com.

Related Work in Mashups

Many efforts try to give users simple mashup tooling for assembling a Web application that integrates different resources. Mashups are mainly data-driven Web applications that combine data from different sources and present them to the user by adding user interface (UI) widgets. No clear separation exists between the data flow and the UI widgets that render that data. Current mashup tools mainly focus on seamless end-user programming without requiring any programming experience from the end users.[1,2] Nevertheless, it's unclear to what extend such end-user mashup tools will allow users to create reuseable and enterprise-scale mashups. Google Mashup Editor (http://editor.googlemashups.com) and Yahoo Pipes (http://pipes.yahoo.com) are two popular Web-based mashup editors for data-driven composition (mainly returning RSS and Atom feeds). Both use proprietary formats for representing and storing the mashup logic. QEDWiki (http://services.alphaworks.ibm.com/qedwiki) is another mashup tool that lets users create UI widgets that access data from various services. Additionally, QEDWiki adopts wikis' idea to create mashups collaboratively. In contrast to mashups and the aforementioned tools, Bite features an integrated programming model for building Web-scale workflows, enabling users to specify control flow, data flow, and flexible UI integration capabilities that aren't narrowed to a specific type of application domain (as mashups are).

E. Michael Maximilien and his colleagues present another interesting approach for creating mashups.[3] They developed a domain-specific language (DSL) for creating mashups called *Swashup*. The DSL is implemented in Ruby on Rails and provides concepts that let users efficiently create mashups: multiprotocol service and data access, mediation support, and a means for generating a UI for the resulting mashup. In contrast to Bite, Swashup doesn't provide a lightweight composition support acting as a foundation for implementing workflows that can integrate UI on top of it. Thus, Bite is well-suited for building data-driven applications (the typical mashups case) but also for implementing workflows with rich execution semantics.

References

1. J. Wong and J.I. Hong, "Making Mashups with Marmite: Towards End-User Programming for the Web," *Proc Int'l Conf. Human Factors in Computing Systems* (CHI 08), ACM Press, 2007, pp. 1435–1444.
2. R. Tuchinda et al., "Building Mashups by Example," *Proc 5th Int'l Conf. Intelligent User Interfaces* (IUI 08), ACM Press, 2008, pp. 139–148.
3. E.M. Maximilien et al., "A Domain-Specific Language for Web APIs and Services Mashups," *Proc 5th Int'l Conf. Service-Oriented Computing* (ICSOC 07), Springer-Verlag, 2007, pp. 13–26.

Rania Khalaf is a research staff member in the Component Systems Group at IBM T.J. Watson Research Center. Her research interests include component-based software engineering, workflow, service-oriented computing, and Web services. She has a PhD in computer science from the University of Stuttgart. Contact her at rkhalaf@us.ibm.com.