On Composing RESTful Services

Cesare Pautasso

Faculty of Informatics
University of Lugano (USI)
via Buffi 13
CH-6900 Lugano, Switzerland
+41 058 666 4311
`c.pautasso@ieee.org`
`http://www.pautasso.info/`

Abstract. Composition is one of the central tenets of service oriented computing. This paper discusses how composition can be applied to RESTful services in order to foster their reuse. Given the specific constraints of the REST architectural style, a number of challenges for current service composition languages and technologies are identified to point out future research directions.

1 Introduction

How do composition and the REST architectural style [1] fit together? What is a composite RESTful service[1]? What is the difference between a mashup and a service composed out of RESTful Web service APIs? The goal of this paper is to collect some answers to these questions by summarizing the discussions during an open break-out session held at a recent Dagstuhl seminar on Software Service Engineering [3]. In particular, we aim at distilling some research challenges and problems that emerge when service oriented architectures made out of composite services are built using REST.

2 Background

Composition is one of the core principles of service-oriented computing [4]. It fosters the reuse of existing services by means of assembling them in multiple applications that combine them in novel and unexpected ways. This principle is not explicitly found in the original definition of the REST architectural style [5].

Instead, the REST architectural style introduces a set of architectural elements (user agents, proxies, gateways, and origin servers) that are meant to be combined to build a layered and scalable system, which enables a large number of clients (or user agents) to access the resources published by a single origin server (Fig. 1). Each element is connected using the HTTP protocol.

[1] An introduction to non-composite RESTful Web services can be found in [2].

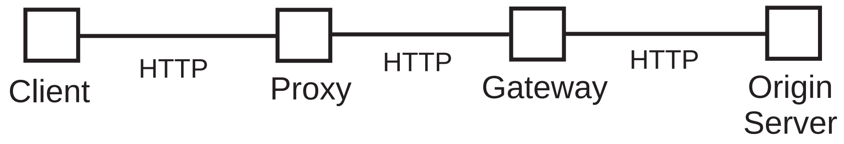

Fig. 1. REST basic architectural elements and connectors

Intermediate components (i.e., proxies and gateways) are optional and are usually added to an architecture compliant with the REST style to perform access control, caching, and some kind of protocol translation. For example, as shown in Fig. 2 a caching reverse proxy (or gateway) may be introduced to reduce the load on the origin server imposed by a growing number of clients. Likewise, an access control proxy may be connected to multiple user agents and only allow a subset of them to access the origin server. For both examples, each client request as it goes through the intermediate element is serviced independently by one origin server.

Whereas multiple servers can exist, they are seen as autonomous and disconnected sources of information whose state evolves independently. Clients may sequentially access multiple servers (e.g., as they follow hyperlinks from one to the other). While doing so, clients may also somehow collect the information originating from multiple servers and aggregate it locally. Thus, REST would seem to support some form of composition *limited to the client*. Still, no intermediate element which can aggregate information from multiple servers is explicitly foreseen.

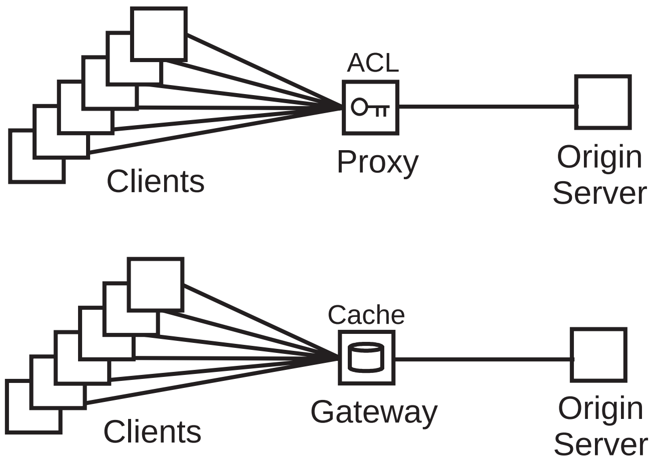

Fig. 2. Access control and caching layers

3 What is a Composite RESTful Service?

In the context of the previous discussion, a composite RESTful service is a special kind of intermediate element which – unlike proxies and gateways – does not simply forward requests to upstream origin servers but may decompose a request so that it can be serviced by invoking more than one origin server (Fig. 3).

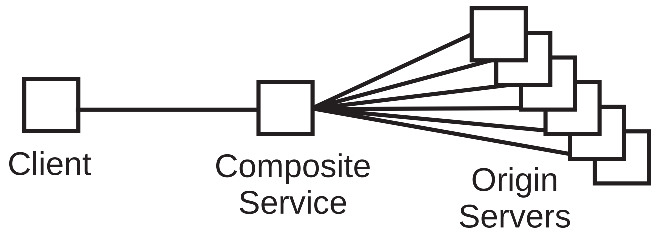

Fig. 3. Composite RESTful service

Given the recursive nature of composition [6], also the composite service is a RESTful service. Thus, it publishes a set of resources to its clients, which interact with them following the REST uniform interface principle (i.e., using the `GET`, `POST`, `PUT`, `DELETE` methods). Moreover, clients do so without knowing that their requests are serviced by a composite service (which acts similar to any other intermediate element).

Thus, clients may not know whether the result of a `GET` request is computed locally by the composite service or it results from combining the results of a set of `GET` requests sent to the composed origin servers. Likewise, as clients initialize (`POST`), update (`PUT`), or delete (`DELETE`) the state of the composite service, their requests may result in resources being created, updated, or deleted locally, or in similar actions being performed by the composite service on a subset of the resources published by the composed origin servers.

From an implementation perspective, we can distinguish two kinds of composite services: *stateless* compositions and *stateful* compositions. Stateful compositions augment the state of the composed origin servers with information that is managed and stored on the composite service. Thus, part of the state of the composite resources is managed locally and the rest is partitioned among the origin servers. Stateless compositions instead do not maintain any local state and transparently map all clients requests to the state of the origin servers.

3.1 Mashups vs. Composite RESTful Services

The above definition may be contrasted with existing work on Web 2.0 mashups [7,8,9] as follows. Mashups can be seen as composition applied at the UI/presentation layer [10], where a fully integrated user interface is built out of reusable widgets applied to data sources. Data sources may provide data “willingly”, e.g., through an ATOM/RSS feed. In some cases data may have to be extracted by reconstructing a so-called scrAPI using screen scraping techniques [11]. Thus, mashups over RESTful services do not emphasize the reusability of the composition, but only the ease with which it can be built [12]. Composite RESTful services instead are meant to be primarily reused as a service, since they do not necessarily aim at providing a user interface. Still, nothing prevents a composite RESTful service from using HTML as one of the representation formats for its composite resources, thus enabling the composite service to be accessed via a fully integrated, mashup-like user interface running in a Web browser [13].

Another difference lies in the notion that most mashups perform some kind of read-only, multi-source data aggregation, whereas composite RESTful services are more general as they allow both read-only access together with the possibility of transferring state from the client to update the composed resources as well as to transfer state laterally among the composed origin servers (i.e., by using the results of a `GET` request performed on one to initialize the state of a new resource on the other using `POST`).

Conversely, mashups and RESTful service composition can also be seen as being complementary. Pure client-side mashups – due to the single origin security policy of most common Web browsers – are very limited in the number of different origin servers that they may contact (i.e., only one). A composite RESTful service would thus complement the integrated user interface of the mashup by helping it to outsource its composition logic and deploy it where it can access multiple origin servers without restrictions.

4 Challenges

Composing RESTful services – as opposed to traditional WS-* services – presents a set of additional specific challenges and opportunities, briefly outlined in this section.

Missing Interface Description RESTful services do not provide an explicit, machine-readable interface description. To support client (and composite services) developers, these services mostly rely on HTML (or PDF) documentation and on the ease with which the functionality of such services can be interactively discovered from a Web browser. Thus, existing languages that require such machine processable metadata to be available at design-time may not be applicable directly. In practice, a RESTful service may be described using the HTTP Binding available with WSDL 2.0. However, currently such description has to be written by the developer of the composition as it is not usually provided by the service provider.

Uniform Interface RESTful services comply with the uniform interface principle, where resources are manipulated using the `GET`, `PUT`, `DELETE`, and `POST` methods. Not only a composite service should be able to invoke its component services using such primitives, but also it should be able to handle these requests performed by its clients on the published composite resources.

Idempotency Whereas the idempotency featured by the `GET`, `PUT`, `DELETE` methods is well established among the basic REST architectural elements of Fig. 1, it remains to be seen how this property can be translated to a composite service.

Dynamic Late Binding REST relies on the Hypermedia As The Engine Of Application State (HATEOAS) principle, where hyperlinks are used to describe and dynamically discover service interaction protocols. As a consequence, resource URIs to be consumed (and provided) by a composite service may only become known at run time. Languages for RESTful service composition should therefore provide mechanisms for URI generation, extraction, parsing, and binding.

State Inspection Clients should be able to refer to (or "bookmark") the state of a running composition using the hyperlinks it provides to the clients. Additionally, such state may be manipulated by clients that should be able to find out what their options are at every step.

Request correlation There are REST-compliant techniques (such as URI addressing and HTTP cookies) that can be used to correlate client requests with the corresponding state of the published composite resources. Can correlation become simpler to implement for composite RESTful services?

Caching Part of the state of a composition may simply be a cached version of a view over the state of the composed services. Existing caching techniques and mechanisms used with proxies and gateways should be applicable also in this case to help tuning the performance of the composition. The challenge lies in whether it is possible to control such caching declaratively, to hide the complexity of the low level HTTP caching control headers.

Dynamic Typing The representation of resources may not be known at design-time, making some form of dynamic typing necessary to deal with data retrieved from RESTful APIs.

Content Type Negotiation Since resources can have multiple representation formats, compositions should be able to negotiate the most appropriate one with the composed origin servers and also be able to provide the most suitable representation to fit the needs of their clients. This can be implemented by relying upon standard HTTP headers (i.e., `Accept`, and `Content-Type`), but also higher-level constructs could be introduced.

Verification and Testing Considering the dynamic approach to composition fostered by REST, it may become difficult to statically verify properties of a composition at design-time. Conversely, testing at run-time becomes critical to enforce assumptions and check the quality of the composite service while facing the independent evolution of the underlying composed RESTful services.

Exception Handling A composite service should be able to leverage the rich set of standardized error and status codes provided by the HTTP protocol. When applicable, errors from the origin servers should be propagated upstream, while in other cases these should be masked accordingly. Also, invalid client requests should be caught without propagating them on to the composed services. The challenge in this case is to define the appropriate exception handling policies.

Hybrid Composition Modern service composition languages should provide seamless support for both WS-* services and RESTful services. As an example, the BPEL for REST [14] extension allows to natively compose both kinds of services.

Workflow-based Composition Given that many composition languages for Web services are based on the concept of workflow, the question (raised by [15] and further developed in [16]) whether a business process model can be used to implement the logic behind a composite RESTful service is highly relevant and should be explored further.

5 Conclusion

RESTful services are currently seen as a lightweight tool for point-to-point integration betwen service providers and a large number of clients. This paper defines the notion of composite RESTful services and discusses some of the specific challenges involved in composing RESTful services (as opposed to WS-* Web services). These are mostly related to the lack of an explicit, machine-processable interface description and the emphasis on dynamic aspects (e.g., content-type negotiation, late binding, state inspection) of the composition. Addressing them will require to revisit some of the assumptions made by current service composition languages and to devise novel languages and techniques helping to more effectively build composite RESTful services.

Acknowledgements

The author would like to thank the organizers and several participants of a recent Dagstuhl seminar on Software Service Engineering for their feedback on the concept of RESTful service composition. This work is partially funded by the SNSF Project No. 125337 (CLAVOS).

References

1. Fielding, R., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology **2**(2) (2002) 115–150
2. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (May 2007)
3. van den Heuvel, W.J., Zimmermann, O., Leymann, F., Shan, T.: Dagstuhl seminar 09021: Software service engineering. executive summary. (January 2009)
4. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall (2005)
5. Fielding, R.: Architectural Styles and The Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
6. Assmann, U.: Invasive Software Composition. Springer (2003)
7. Maximilien, M., Nielsen, D., Tai, S., eds.: 1st International Workshop on Web APIs and Services Mashups. (September 2007)

8. Wikipedia: Mashup (web application hybrid). `http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`
9. Erenkrantz, J.R., Gorlick, M.M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of Web architectures. In: Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE 2007), Dubrovnik, Croatia (September 2007) 255–264
10. Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., Casati, F.: Understanding UI integration: A survey of problems, technologies, and opportunities. IEEE Internet Computing **11**(3) (May-June 2007) 59–66
11. Schrenk, M.: Webbots, Spiders, and Screen Scrapers. No Starch Press (2007)
12. Vinoski, S.: Serendipitous reuse. IEEE Internet Computing **12**(1) (2008) 84–87
13. Pautasso, C.: Composing RESTful services with JOpera. In: Proc. of the International Conference on Software Composition (SC09), Zurich, Switzerland (July 2009) 142–159
14. Pautasso, C.: BPEL for REST. In: Proc. of the 7th International Conference on Business Process Management (BPM08), Milan, Italy (September 2008) 278–293
15. Overdick, H.: Towards resource-oriented BPEL. In: 2nd ECOWS Workshop on Emerging Web Services Technology. (November 2007)
16. Rosenberg, F., Curbera, F., Duftler, M.J., Kahalf, R.: Composing RESTful services and collaborative workflows. IEEE Internet Computing **12**(5) (September-October 2008) 24–31