Chapter 12
Service-Driven Computing with APIs: Concepts, Frameworks, and Emerging Trends

Hiranya Jayathilaka
University of California – Santa Barbara, USA

Chandra Krintz
University of California – Santa Barbara, USA

Rich Wolski
University of California – Santa Barbara, USA

ABSTRACT

While both SOAP and REST have been used widely to implement Web services and software integration, over time REST has emerged as the predominant approach. REST provides developers with a lower barrier to entry for implementation and greater development flexibility than SOAP. Its architectural conventions and best practices can be integrated into Web services incrementally as opposed to the all-or-nothing adoption of SOAP. In order to achieve generality, SOAP standards are extensive, rigid, and complex. This complexity can lead to implementations that introduce significant overhead on the network bandwidth consumption, execution times, and throughput of SOAP services, especially in the emerging resource-restricted mobile realm. This chapter provides an overview of the logical and physical design of modern Web services and discusses the strengths and weaknesses of the predominant styles. It provides evidence and reasoning behind the emergence of REST as the leader for the development of next-generation Web APIs and services. The chapter also delineates the key technologies that underlie REST and describes emerging and future research directions in support of REST-based APIs and service development.

DOI: 10.4018/978-1-4666-8619-9.ch012

INTRODUCTION

Web services (W3C, 2004b) simplify the development of network-accessible distributed applications by combining the ubiquity of the Internet with familiar protocols of the World Wide Web (WWW), well-defined interfaces, and easily integrated software architectures. They expose functionality, business logic, and data as network-enabled modules, which can be consumed over the network by client applications written in a variety of programming languages. The *Web service* design and development model which utilizes *Service Oriented Architecture* (SOA), describes how to architect, reuse, and integrate Web service modules as the building blocks for new systems and services (Dan, Johnson, & Carrato, 2008). SOA is used increasingly for a wide range of application domains including business collaboration and productivity, Web/mobile access and communications, large scale data integration and analysis, multi-player games, and Cloud computing (compute resources, software, and data "as-a-service") (Haines & Haseman, 2009). Many industry technology leaders (e.g., Google, Yahoo, Amazon, eBay, and Facebook) use SOA as the basis of their products, which they expose to external developers for integration with new products, services, and platforms.

Systems implemented using Web services and SOA principles tend to be loosely coupled and resilient to change, failure, and interruption (Offermann, Hoffmann, & Bub, 2009). The modular nature of Web services promotes separation of concerns, and makes it easier to design and reason about distributed applications. In addition, Web service encapsulation enables developers to choose the most appropriate programming language and technologies for their implementations and to isolate change across complex systems. This service-driven approach for developing applications minimizes code duplication and eases the assembly of complex systems, thereby greatly improving developer productivity over non-service-oriented methodologies. By composing an application from existing services that encapsulate common tasks such as database access, logging, and security; application developers can work at a higher level of abstraction, thereby saving development and debugging time. In particular, the reuse of services as modules increases reliability and stability in the resulting software system, since testing and quality assurance can focus on integration rather than on the constituent services.

Logically, a Web service consists of four primary components:

- An *architecture* that governs the logical organization of data, code, and communication of the service.
- *Abstraction* that hides the implementation details of the service and that enables the service architect to control the functionality that is exposed to users and other programs (service clients).
- An *implementation* that contains program code for computation and data manipulation that is executed when a client accesses the service.
- An *application-programming interface* (API) that defines and controls the operations that clients perform to access the implementation as specified by the abstraction layer.

The architecture and abstraction are the logical constructs of Web service design and the API and implementation comprise its physical manifestation. From an implementation perspective, APIs decouple the functionality necessary to allow controlled access to a service from the technologies that are used to implement the functionality of the service itself (Beyer, Chakrabarti, & Henzinger, 2005). That is, the API preserves the service functionality for clients accessing the service while the technologies used to implement it can change, particularly as technological advances reduce implementation costs. Similarly, it is pos-