Cesare Pautasso · Erik Wilde
Rosa Alarcon *Editors*

REST: Advanced Research Topics and Practical Applications

Springer

REST: Advanced Research Topics and Practical Applications

Cesare Pautasso • Erik Wilde • Rosa Alarcon
Editors

REST: Advanced Research Topics and Practical Applications

Springer

Editors
Cesare Pautasso
Faculty of Informatics
University of Lugano
Lugano
Switzerland

Erik Wilde
EMC Corporation
Pleasanton, California
USA

Rosa Alarcon
Pontificia Universidad Catolica de Chile
Santiago
Chile

ISBN 978-1-4614-9298-6 ISBN 978-1-4614-9299-3 (eBook)
DOI 10.1007/978-1-4614-9299-3
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2013951522

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Contributors

Timo Aaltonen Tampere University of Technology, Tampere, Finland

Rosa Alarcon Department of Computer Science, Pontificia Universidad Catolica de Chile, Santiago, Chile

Mike Amundsen Principal API Architect for Layer 7 Technologies, Washington, USA

Luis Cipriani Abril Midia Digital, São Paulo, Brasil

Craig Dowie Betfair, London, United Kingdom

Renata Pontin de Mattos Fortes Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brazil

Morteza Ghandehari Department of Computing Science, University of Alberta, Alberta, Canada

Michael M. Gorlick University of California, Irvine, CA, USA

Andreas Harth Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Michael Hausenblas DERI, Galway, Ireland

Markus Lanthaler Institute for Information Systems and Computer Media, Graz University of Technology, Graz, Austria

Myriam Leggieri Digital Enterprise Research Institute (DERI), National University of Ireland, Galway, Ireland

Maria Maleshkova Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Ricardo Ramos de Oliveira Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brazil

Cesare Pautasso University of Lugano, Lugano, Switzerland

Luiz Rocha Abril Midia Digital, São Paulo, Brasil

Robson Vincius Vieira Sanchez Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brazil

Steffen Stadtmüller Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Thomas Steiner Departament de Llenguatges i Sistemes Informatics, Universitat Politècnica de Catalunya, Barcelona, Spain

Vlad Stirbu Tampere, Finland

Eleni Stroulia Department of Computing Science, University of Alberta, Alberta, Canada

Mohsen Taheriyan Information Science Institute, University of Southern California, Marina del Rey, CA, USA

Richard N. Taylor University of California, Irvine, CA, USA

Ruben Verborgh Multimedia Lab – Ghent University – iMinds, Ledeberg-Ghent Belgium

Rik Van de Walle Multimedia Lab – Ghent University – iMinds, Ledeberg- Ghent, Belgium

Marcel Weiher Metaobject Ltd., London, United Kingdom

Erik Wilde EMC Corporation, Pleasanton, CA, USA

Editorial Board

Chapter 1
Introduction

Cesare Pautasso, Erik Wilde and Rosa Alarcon

Representational State Transfer (REST) is an architectural style that defines the architectural quality attributes of the World Wide Web, seen as an open, loosely coupled, massively distributed and decentralized hypermedia system. REST has seen a large uptake for the last several years, as it is largely regarded as a simpler and more Web-like way of exposing service interfaces, in particular when compared with earlier approaches such as the rather complex and heavyweight SOAP/WS-* and similar RPC-inspired protocols. As with all success stories, there also is the downside that pretty much any HTTP-based service is being promoted and sold as being "RESTful" these days, even though many do not completely follow the architectural principles and constraints underlying the REST style [81]. However, in the end this may be the fate of every successful "brand name": once it becomes successful enough, it loses its purity and is used for marketing, where technical accuracy is not the main concern.

The goal of this book is to present the latest developments and advances in research around the topic of REST, and how the Web is changing from a distributed hypermedia system for document publishing to a programmable medium which is more and more being used to deliver software as a service. We have collected many important contributions exending the scope of applicability of REST, proposing to update REST to fit with modern requirements, discussing how to model and describe RESTful architectures, as well as dealing with some of its limitations (e.g., observing

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

C. Pautasso (✉)
University of Lugano, via Buffi 13, 6900, Lugano, Switzerland
e-mail: c.pautasso@ieee.org

E. Wilde
EMC Corporation, 6701 Koll Center Parkway, Pleasanton, CA 94566, USA
e-mail: erik.wilde@emc.com

R. Alarcon
Department of Computer Science, Pontificia Universidad Catolica de Chile,
Casilla 306, Cod. 143, Santiago 22, Santiago, Chile
e-mail: ralarcon@ing.puc.cl

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_1,

event notifications). These forward-looking chapters are complemented by three case studies, giving a practical perspective on current successful applications of REST in the real world.

1.1 REST Design Constraints

The REST architectural style has been derived from a set of well-know styles (e.g., layered, client/server); it fosters high performance and scalability by facilitating replication on the server-side and on the intermediary layers (i.e., caches provide temporary, on-demand replicas). Ensuring the statelessness of the client-server interaction simplifies replication, and contributes to a more reliable system where failures can be handled gracefully. Concerns are separated in layers known as the *origin server*, the *user agent* (or browser), and a set of *intermediaries* (e.g. proxies, reverse proxies, etc.). This approach not only facilitates flexibility and scalability, but also it makes possible to perform intermediary processing of messages, facilitating the administration of some quality capabilities such as performance (e.g., data compression or caches) or security (e.g., authentication and access control). The code-on-demand characteristic allows to extend the browser functionality, and to personalize Web applications according to a particular user's needs or device constraints.

The main characteristic that sets REST apart from other architectural styles is the definition of a *uniform interface* to be shared among all architectural components. It requires that data elements (*Web resources*) are identified through global and unique addresses or identifiers (e.g., URIs). These identifiers are globally meaningful, so that no central authority is involved in minting them, and they can be dereferenced independently of any context. The resource itself is a conceptual entity, and REST does not make any assumptions on the corresponding implementation. The uniform interface requires also that resource state can be retrieved through a representation. Representations must be used also to manipulate the resource state, by sending requests that represent intended state changes. The uniform interface constraints requires also self-descriptive messages; client and servers interact with each other by exchanging request and response messages, which contain both the data (or the representations of resources in a specific serialization format) and the corresponding metadata (information about the resource representation). Representation formats can be negotiated and vary according to the client context, interests, and capabilities. For example, a mobile client can retrieve a low-bandwidth representation of a resource. Request and response messages metadata makes possible that services do not need to assume any kind of out-of-band agreement on how the representation should be parsed, processed, and understood.

Messages are subject to a network protocol that indicates how they should be processed. For instance, in case of the HTTP protocol, a set of methods (e.g., GET, PUT, DELETE, POST, HEAD, OPTIONS, etc.) with well defined semantics in terms of the effect on the state of the resource, are defined. HTTP methods can be applied to all Web resource identifiers (e.g., URIs which conform to the HTTP scheme). The

set of methods can be extended if necessary (e.g., HTTP PATCH has been recently standardized as an addition to deal with partial resource updates), and other protocols based on HTTP such as WebDAV include additional methods. Responses in HTTP have also clear semantics conveyed by the metadata associated and a set of status codes (e.g. 2xx successful processing, 3xx redirection, 4xx failure caused by the client, 5xx failure caused by the server).

Finally, the uniform interface introduces the so called *hypermedia* constraint as a mechanism for decentralized resource discovery by referral. Hypermedia is about embedding references to related resources inside resource representations or in the corresponding metadata. Clients can thus discover the identifiers (through hyperlinks) of related resources when processing representations, and choose to follow these links as they navigate the graph built out of relationships between resources. Hypermedia helps to deal with decentralized resource discovery, and is also used for dynamic discovery and description of interaction protocols between services. Despite its usefulness, it is also the constraint that has been the least used in most Web service APIs claiming to be RESTful. Thus, sometimes Web service APIs which also comply with this constraint are also named "Hypermedia APIs". In addition to links, hypermedia *controls* (such as forms allowing to POST content on the Web) allow to manipulate and change the state of resources.

1.2 REST Activities and this Book

Starting in 2010, the editors of the book you are currently reading started organizing a series of workshops under the label of "WS-REST", which on the one hand stands for "Workshop for REST", but also is a play on the plentiful WS-* standards that appeared in rapid succession before that time. The WS-REST workshops were always co-located with the popular WWW conference, and were held in the years [5, 182, 183] and [6]. In 2011, the first two workshops were used as a basis for a first book, which was published in late 2011 as "REST: From Research to Practice" [250].

This second book, "REST: Advanced Research Topics and Practical Applications", combines a second interesting set of REST-related papers, structured into two parts about "REST Research" and "Practical Applications" respectively. We trust that this second book will provide readers with the same interesting and inspiring perspectives on REST.

1.2.1 REST Research

Part I *REST Research* focuses on research work that is looking into specific aspects of the REST architectural style.

Chapter 2, "Communication and Capability URLs in COAST-based Decentralized Services", describes COAST and one of the specific aspects of this approach. COAST builds upon CREST, which itself was conceived as an extension of the basic REST

style. Both COAST and CREST are attempts to take REST's resource-centric architecture, and extend it with computational concepts. The goal of COAST and the specific concepts presented in this chapter is to present an architecture that works well for decentralized and independently evolving service scenarios.

Chapter 3, "Interoperability of two RESTful protocols: HTTP and CoAP", looks at how to combine the predominant way of doing REST, which is HTTP over TCP, with another RESTful protocol, which is CoAP over UDP. The scenarios for this interworking are from the Internet of Things and Web of Things, where communications with embedded and resource-constrained devices often cannot afford to use HTTP's full protocol stack, and yet want to fit into a REST framework so that the overall architecture is still loosely coupled.

Chapter 4, "Enabling Real-Time Resource Oriented Architectures with REST Observers", discusses one of the challenges that are both a fundamental constraint of the basic REST approach, and yet need to be addressed for certain scenarios. The problem is REST's client-driven interactions, which mean that it's not entirely natural to model systems where state changes in resources should be communicated to clients. The REST observers introduced in this chapter are one pattern to solve this problem.

Chapter 5 is a "Survey of Semantic Description of REST APIs", which looks at the variety of formal and semi-formal description approaches that have been layered on top of REST's basic principles. The survey is structured into lightweight semantic descriptions, graph patterns, logic-based descriptions, JSON-based descriptions, and annotation tools for creating descriptions.

Chapter 6 presents "APIs to Affordances: A New Paradigm for Services on the Web", and looks at the ways in which REST as a hypermedia-oriented style allows designers and providers of services to focus on network-oriented affordances instead of device-oriented APIs. The chapter represents a "what-if" proposal; an alternate paradigm for dealing with an increasingly growing and heterogeneous network. Drawing from diverse sources including physical architecture, industrial design, the psychology of perception, and cross-cultural mono-myth, a new implementation paradigm is proposed to help software architects and developers meet these challenges.

Chapter 7 discusses "Leveraging Linked Data to Build Hypermedia-Driven Web APIs", by first introducing JSON-LD as a bridge between Semantic Web technologies and easily processable JSON representations, and then presenting Hydra, a lightweight vocabulary for hypermedia-driven Web APIs. Similar to the approaches presented in Chapter 5, also Hydra uses machine-readable annotations.

Chapter 8 discusses "RestML: Modeling RESTful Web Services", and thus is in a similar space as the previous chapter, which also discusses how to represent information that helps to describe REST services. However, Chapter 8 takes an approach based on UML, presenting a language that is called RestML. In contrast to Hydra, which is mostly intended to help understanding REST services, RestML takes an approach that targets code generation, so that REST services can be used in model-driven approaches.

1.2.2 Practical Applications

Part II *Practical Applications* looks at applications of REST as an architectural style, and presents three different case studies in particular application areas.

Chapter 9 presents "A Lightweight Coordination Approach For Resource-Centric Collaborations", a coordination approach and a supporting framework for resource-centric collaborations. The presented solution consists of a language and tool support for specifying collaborative activities and the resources they manipulate, an engine for enacting them at run time, and a systematic methodology for integrating the engine with the various interactive systems and services involved.

Chapter 10 is about "Connecting the Dots: Using REST and Hypermedia to publish Digital Content", and presents the design and implementation experience of a REST-based platform for a large publisher. The Alexandria platform is a system of systems, distributed, decentralized but interconnected, that allows each component to evolve independently and the platform to grow organically as support for more business needs is added.

Chapter 11 describes "In-Process REST at the BBC" as a way to show how REST is used more on the software engineering level than on the information system engineering level. It describes how the BBC feed processing platform for transforming structured XML information of live sporting events to HTML output for the BBC website in soft real time was redesigned and reimplemented following REST principles, resulting in a simpler system design with significant performance improvements.

Part I
REST Research

Chapter 2
Communication and Capability URLs in COAST-based Decentralized Services

Michael M. Gorlick and Richard N. Taylor

2.1 Introduction

Decentralized software systems are distributed systems that span multiple, distinct spheres of authority—participants may unilaterally change their behaviors in ways that may or may not be compatible with the needs or goals of the other members. The web is a prime example; servers come and go, links are created and broken, and mashups are deployed that rely upon the APIs of other web sites. Integrated supply chains are another example; the designs of NASA's Curiosity rover and both the Boeing 787 and Airbus A380 commercial aircraft required the network-mediated collaboration of thousands of engineers in many dozens of companies. Decentralization appears in such varied domains as disaster response, coalition military command, commerce, finance, education, and scientific research. A decentralized system may be open or closed. In the former participation is loosely constrained if at all, while in the latter participation is governed by agreements (with varying degrees of formality, rigor, and enforcement) among the participants. The global web is an open system and anyone can participate, but participating in a business-to-business supply chain system demands negotiations and contracts. Joining or leaving the global web can be done on a whim, but joining or leaving a supply-chain system should not be undertaken lightly.

All decentralized systems are intrinsically dynamic: members join and leave, service relationships change, system implementations and deployments vary (as do their rates of evolution and adaptation), and members adapt to the changing business, financial, or regulatory environment. Both open and closed decentralized systems raise concerns of security and trust and neither is immune to malicious behavior.

---

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

M. M. Gorlick (✉) · R. N. Taylor
University of California, Irvine, CA 92697-3455, USA
e-mail: mgorlick@acm.org

R. N. Taylor
e-mail: taylor@ics.uci.edu

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_2,

COmputAtional State Transfer (COAST), an architectural style based on the idiom of *computation exchange*, targets decentralized systems and their associated security issues. COAST has its roots in two earlier architectural styles, REST and CREST. The World Wide Web is one of the best known decentralized applications and REST (Representational State Transfer) is the architectural style [81] underlying the Web's evolution, performance, and scaling. Code mobility was always part of the REST style (for example, Javascript embedded in HTML pages) with the nominal goals of fostering browser-side display of new media types or reducing application latency. In other words, computation mobility in REST was subservient to content transfer and focused largely on optimizing the transfer and interpretation of resource representations.

On one hand REST was a huge success, as adherence to the REST principles set the stage for the Web's unparalleled expansion. However, REST has many shortcomings. From the outset there was insufficient support for differentiation, as the rapid adoption of cookies containing keys to session state held server-side, in violation of REST precepts, demonstrated. In contrast to cookies, web continuations [132] preserve stateless interactions. The emergence of Ajax (mashups) and the exploitation of computation in the browser suggested a more prominent role for code mobility [89], namely constructing and deploying customizations and application services [65]. At the same time inadequate security led to numerous breaches.

Inspired by REST, the evolution of web architecture, and the rapid introduction of Ajax and Web Services, we formulated CREST [65–67], an architectural style in which computations displaced content representations as the unit of exchange among hosts. In CREST, actively executing computations (as opposed to "resources" as abstracted black-boxes of information) were named by URLs and computations exchanged state representations reified as closures and continuations. Our trials of CREST, including a customizable, collaborative feed reader and analyzer [66, 69] and Firewatch [95], a system for wildfire detection and response, showed considerable promise for constructing highly dynamic systems. However, CREST needlessly inherited many constraints from Web architecture and, like REST before it, failed to address security in any comprehensive manner.

COAST, the successor to CREST, is a style for which security is a dominant concern and whose mechanisms allow hosts to minimize the risk of executing visiting computations on behalf of clients. A detailed view of COAST accompanied by a demonstration application is given in [97]. Here our focus is communication security, whereby COAST hosts manage communications among computations and modulate access to critical services. However, before introducing these communication mechanisms we describe our domain of interest, decentralized SOAs, from the perspective of computation exchange and from there move to the COAST style itself. With that behind us we turn to our principal contribution, the details of Capability URLs, and present examples of their use.

2.2 Decentralized Systems via Computation Exchange

Decentralized systems whose constituent subsystems operate under distinct spans of authority must meet two conflicting goals: protecting valuable fixed assets (such as servers, databases, sensors, data streams, and algorithms) and meeting the evolving service demands of a diverse client population.

Computation exchange (the computational analogue of content exchange) is the bilateral exchange of computations among decentralized peers. In this regime, content delivery is a by-product of the evaluation of computations exchanged among peers. Computation exchange exploits existing core organizational functions, processes, and assets to create higher-level customized services, but imposes significant security obligations.

Computation exchange generalizes and subsumes a number of well-known styles for distributed computing, including remote procedure call [32, 164], remote evaluation [219–221], REST [81], and service-oriented architectures [70]. From the perspective of computation exchange remote procedure call is an exchange containing a single function call, remote evaluation is an exchange containing an entire function body, REST is an exchange of a small set (GET, PUT, POST, DELETE, and so on) of single function calls accompanied by call-specific metadata, and service-oriented architectures are higher-order compositions of remote evaluation.

Computation exchange induces all of the risks associated with mobile code [89] including waste of fungible resources (processor cycles, memory, storage, or network bandwidth), denial of service via resource exhaustion, service hijacking for attacks elsewhere, accidental or deliberate misuse of service functions, or direct attacks against the service itself. A computation accepted from a trusted source may be erroneous or misapply a service function due to honest misunderstanding or ambiguity. Even a correct computation may expose previously-unknown bugs in critical functions, leading to inadvertent loss of service.

For decentralized systems authentication, secrecy, and integrity are necessary but insufficient for asset protection as there is no common defendable security perimeter when function is integrated across the multiple, separate trust domains [257]. Here an attack on one authority threatens all. At best a breach may lead to failures in other trust domains. At worst a breached authority may undertake an "insider" attack against its confederates. With this in mind decentralization demands that security be everywhere always. Applications that cross authority boundaries inherently bring security risks; adaptations in such contexts only increase the peril, hinting that security should be a core *architectural* element.

2.3 The COAST Architectural Style

COmputAtional State Transfer (COAST) is an architectural style for decentralized and adaptive systems [97]. Its applications have origins in CREST and, before that, the REST architectural style. COAST targets decentralized applications where organizations offer execution hosts (called *islands*) whose base assets can include

databases, sensors, devices, execution engines, domain-specific functions, or access to distinctive classes of users. In COAST, third-party organizations create their own custom-tailored versions of services (modulo the constraints imposed by the asset owner) by dispatching computations to asset-bearing islands. For instance, a monitor-and-alert function may be defined by one user to run periodically on an island offering access to a collection of environmental sensors. Mobile code both implements the computations in a COAST system and defines the messages exchanged among those computations. Decentralized security and guarding against untrusted or malicious mobile code are principal island concerns—the style mandates architectural elements that when used appropriately provide access, resource, functional, and communication security. In exchange for the complexity imposed by these security mechanisms, COAST allows the construction of on-demand tailored services and enables a wide range of dynamic adaptations in decentralized systems.

COAST security relies on the *Principle of Least Authority* (POLA) [202] and *capability-based security* [34]. POLA dictates that security is a product of the authority given to a principal (the functional power made available) and the rights given to the principal (the rights of use conferred with respect to that authority). At each point within a system a principal must be simultaneously confined with respect to both authority and rights. A *capability* is an unforgeable reference whose possession confers both authority and rights of access to a principal. COAST is one architectural style for computation exchange, just as "pipes and filters" is one of many architectural styles for data processing. COAST's constraints mandate where, when, and how authority and rights are conveyed.

The COAST style states:

- All services are computations whose sole means of interaction is the asynchronous messaging of closures (functions plus their lexical-scope bindings), continuations (snapshots of execution state [74]), and binding environments (maps of name/value pairs [113])
- All computations execute within the confines of some execution site $\langle E, B \rangle$ where E is an execution engine and B a binding environment
- All computations are named by Capability URLs (CURLs), an unforgeable, tamper-proof cryptographic structure that conveys the authority to communicate
- Computation x may deliver a *message* (closure, continuation, or binding environment) to computation y only if x holds a CURL u_y of y
- The interpretation of a *message* delivered to computation y via CURL u_y is u_y-dependent

For example, Alice operates a COAST-based high-performance image processing service. Her clients dispatch computations for processing, enhancing, and analyzing a wide variety of commercial, industrial, and scientific imagery to her service. The execution sites in her server farms are managed by her own COAST computations whose CURLs denote site-specific processing varying across a spectrum of performance and functionality.

Bob, whose machine shop manufactures custom aviation and motorcycle racing components, is one of Alice's clients. His COAST-based automated visual inspection

system dispatches quality-control computations containing high-resolution digital photographs of components to Alice's execution sites for final inspection. Alice's proprietary algorithms combined with Bob's customized closures for component- and use-specific analysis help Bob maintain a high level of quality.

Carol, another of Alice's clients, analyzes medical imagery for physicians and medical testing labs. The sheer volume of the imagery, along with strict medical privacy regulations, prevent Carol from shipping her closures, binding environments, and imagery to an outside processor (as Bob does for his custom racing components), so Carol has licensed an image processing library from Alice that has been integrated into the execution sites of her own in-house COAST-based services.

Carol obtains analytical tools for her imagery from Dave, whose biotechnology company deploys computations for narrowly targeted tissue analyses to COAST sites. Carol dispatches service requests (as computations) to Dave's COAST services. Each of her requests prompts Dave's computations to generate a custom analysis (as a closure or continuation) optimized to meet her request-specific needs and constraints. Included in each of Carol's requests is a nondelegable, "use-once-only" CURL referencing one of her execution sites containing privacy-sensitive medical images.

Dave deploys his customized analysis to Carol's site via Carol's CURL. As Dave's analysis executes on Carol's COAST infrastructure her execution site prevents Dave's computation from accessing any other confidential imagery. Her COAST-based monitoring and auditing infrastructure tracks the execution of Dave's analysis from beginning to end, ensuring that it does not violate patient privacy regulations. The nondelegable, use-once-only CURL prevents Dave from sharing the CURL with any other COAST site (nondelegation), and, as it can never be used more than once, neither Dave nor any attacker that infiltrates Dave's infrastructure can ever send more than a single computation to Carol's request-specific, privacy-sensitive execution site. In general, Carol can monitor Dave's computations executing on her infrastructure or her own computations executing elsewhere using automated "report-back" CURLs, selectively wrapped closures, monitoring functions provided at the service execution site, publish/subscribe events originated by a service provider, or third-party monitoring.

COAST offers two distinct forms of capability, (1) functional capability—what a computation may do, and (2) communication capability—when, how, and with whom a computation may communicate. Functional capability is regulated by execution sites while communication capability is regulated by CURLs. These two mechanisms, execution sites and CURLs, can be combined in many different ways to elicit domain- and computation-specific security.

Execution Sites Over its lifespan each COAST computation is confined to an *execution site* $\langle E, B \rangle$. The execution engine E may vary from one execution site (and computation) to another: for example, a Scheme interpreter or a JavaScript just-in-time compiler. The execution engine defines the execution semantics of the computation and the machine-specific limits (e.g., resource caps) imposed upon the computation.

The binding environment B contains all of the functions and global variables offered to the computation at that execution site. Names unresolved within the lexical scope of computation c (the *free variables* of c) are resolved, at time of reference, within the binding environment B. If B fails to resolve the name the computation is terminated.

Both the execution engine and binding environment of an execution site $\langle E, B \rangle$ may vary independently and multiple sites may be offered within a single address space. E may enforce site-specific semantics: for example, limits on the consumption of resources such as processor cycles, memory, storage, or network bandwidth; rate-throttling of the same; logging; or adaptations for debugging. The contents of B may reflect both domain-specific semantics (for example, B contains functions for image processing) and limits on functional capability (B contains functions for access to a *subset* of the tables of a relational database).

Capability URLs CURLs convey the ability to communicate between computations. A CURL u issued by a computation x is an unguessable, unforgeable, tamper-proof reference to x, as it contains cryptographic material identifying x and is signed by x's execution host. A CURL referencing x may be held by one or more other computations y. CURL u is a capability that designates the network address of computation x, contains arbitrary x-specific metadata (including closures), and grants to any computation y holding u the power to transmit messages to x. When y transmits a message m to x via CURL u both the message m and the CURL u are delivered together to x. The CURL u specifies which execution site $\langle E, B \rangle$ of x is to be used for the interpretation of message m where binding environment B strictly confines the functional capability granted to mobile code contained in the incoming message.

A computation x uses the CURLs it issues to constrain its interactions with other computations and to bound the services it offers. The rationale for constraining interaction in this way is based upon security concerns. A computation y, holding a CURL for x, can send arbitrary closures to x in the expectation that x will evaluate those closures in the context of some x-specific execution site $\langle E, B \rangle$. Therefore x must defensively minimize the functional capability that it exposes to visiting closures.

A computation can accumulate communication capability in the form of additional CURLs. For any computation, CURLs conveying additional communication capability are: (1) contained in the closure defining the computation; (2) returned as values by functions invoked; or (3) embedded as values in the messages received.

Constructing COAST Applications A COAST application is constructed from multiple services available at distinct, decentralized execution sites, each of which offers location- and organization-specific primitives. Those services themselves may depend on customized collaborations with yet other services. Fig. 2.1 illustrates the notional structure that COAST induces on execution hosts.

Computations are expressed in MOTILE, a single-assignment functional language with functional, persistent data structures [173] (all data structures are immutable). A COAST *island* is a single, uniform address space occupied by one or more computations. Computations residing on an island I issue one or more CURLs to the

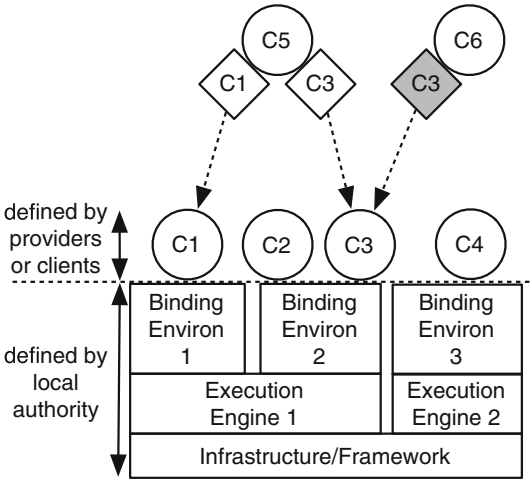


Fig. 2.1 The notional structure of a COAST execution host where a trusted code base allocates execution engines and binding environments to computations, whose implementations are sourced from a variety of other organizations. Distinct functional capability is held by each of the three individual binding environ(ment)s shown here. Computations (shown as *circles*) C1–C4 have been deployed by service providers and clients. Computations C5 and C6 each hold a distinct CURL (*shown as diamonds*) denoting different services offered by computation C3. By holding those CURLs C5 and C6 possess the right (*denoted by dotted arrows*) to dispatch mobile code as messages to C3 for execution

computations with which they wish to communicate. A CURL *u for x* is a CURL generated by x. For the sake of security, communication among computations is "communication by introduction" meaning that computation x can't communicate directly with computation y unless it already holds or obtains (via function call or messaging) a CURL issued by y.

2.4 Capability URLs in Detail

Each CURL u denotes a specific computation x and contains a self-certifying network address [115], a path (a list of MOTILE values), and arbitrary metadata. To ensure the integrity of "introduction only" it must be effectively impossible to guess, forge, or alter a CURL. CURLs are a first-class, immutable capability in MOTILE; hence, within the confines of a legitimate island, it is impossible for a MOTILE computation to forge a CURL or alter one surreptitiously. Every island I holds a public/private key pair and guarantees the integrity of the CURLs that its computations issue by signing each with its private key.

For the sake of safety and security, islands must manage and limit access to both fungible resources (such as memory or bandwidth) and island-specific assets (such as sensors or databases). Restricting the lifespans of computations may help

```
1 (let* ((path (list "question" "ultimate"))
2        (metadata
3         (list (cons "name"      "Arthur_Dent")
4               (cons "residence" "Earth"))))
5       (I@ (curl/new (resource/root) path metadata)))
6   (curl/send
7    Guide@
8    (list "SPAWN" (lambda () (curl/send I@ 42))))
9   (receive))
```

Fig. 2.2 A simple MOTILE program. Island *I* sends a closure to island *Guide* for execution that does nothing but transmit the number 42 back to island *I*

an island stave off resource exhaustion and limiting the total number of messages that a computation may receive or the rate at which they are delivered can limit access, improve performance, or reduce the severity of computation-specific denial of service attacks. These forms of resource security protect against malicious visiting computations intent on resource attacks or exploiting the island as a platform for attacks directed elsewhere.

With these primitive mechanisms at hand it is trivial to generate a "once only" CURL that is invalid after a single use. Finite CURL lifespans allow computations to offer time-limited services to their clients; for example, such CURLs can be used by a transaction coordinator to enforce time limits among the participants of a two-phase commit. An e-commerce service can combine lifespans with use counts to generate the CURL-equivalent of limited-offer coupons or gift cards, and rate limits are useful in "introductory" promotions in which the service may want to bound the rate of use by newcomers.

The CURLs generated by a computation x draw upon a tree of "resource accounts" whose root is the resource account granted to computation x "at birth" by the island I on which x resides. Each account has a finite lifespan and contains a "balance" comprising a use count and rate limit. The initial balance allocated to a new account is "withdrawn" from its parent account and the lifespan of the new account is never more than the lifespan of the parent account. Many accounts may derive from the same parent account and many CURLs may share a single resource account in common.

A Simple Motile Program Fig. 2.2 is an example of a simple program in which a trivial closure is dispatched by island I to a remote island *Guide* for execution and a constant value is transmitted back to island I. Line 5 binds the variable I@ to a CURL for computation x on island I. The function resource/root always returns the root resource account of the calling computation; the MOTILE function curl/new (line 5) generates a CURL given a resource account from which the CURL draws its resources (use count, rate limit, and lifespan), a path (line 1), and metadata (lines 2–4).

The function curl/send given in lines 6–8 transmits a spawn command (the second argument, line 8) to the computation denoted by the first argument, a CURL for island *Guide* bound to the variable Guide@ (line 7; the details of how computation x acquired CURL Guide@ are omitted for the sake of brevity and clarity). The closure, (lambda . . .), evaluated by an execution site of island *Guide*, immediately transmits

```
1  (define (palindrome? s)
2    (let loop
3      ((left 0)
4       (right (sub1 (string-length s))))
5      (or
6       (>= left right)
7       (and
8        (char=? (string-ref s left) (string-ref s right))
9        (loop (add1 left) (sub1 right))))))
10
11 (let* ((reply (promise/new 60.0))
12        (reply/promise (car reply))
13        (reply/curl    (cdr reply))
14        (palindromes
15         (lambda ()
16           (curl/send reply/curl (words/filter palindrome?)))))
17   (curl/send J@ (list "SPAWN" palindromes))
18   (promise/wait reply/promise #f))
```

Fig. 2.3 A computation on island I dispatches a closure to island J to obtain all of the palindromes in a database of words

the message 42 back to computation x on island I via CURL Guide@. The MOTILE function receive, called on line 9 of computation x, blocks until a message m for computation x arrives and returns that message m as its value.

A Client-Defined Service Fig. 2.3 illustrates sending a closure from island I to extract all of the palindromes contained in a database of words maintained by island J. Since island J has no predefined function for detecting palindromes, the computation on island I defines (lines 1–9) a function palindrome? that accepts a string s and returns true (#t) if s is a palindrome and false (#f) otherwise. MOTILE uses promises to bridge the gap between functional programming and asynchronous messaging. A *promise* is a proxy object for a result that is initially unknown because the computation of its value has yet to be initiated or is incomplete. Line 11 creates a new promise with a lifespan of 60 s. In MOTILE a promise consists of two elements: the promise object proper (reply/promise in line 12) and a single-use CURL, reply/curl in line 13, by which the result of the promise will be resolved by some computation .

Lines 14–16 define the closure, palindromes, that will be transmitted to island J for evaluation. palindromes, when executed by island J, first applies the I-defined predicate palindrome? as a filter to the contents of the word database and then sends the result of that filtering (a list, possibly empty, of the palindromic words in the database) to the island computation denoted by the CURL reply/curl. words/filter, a domain-specific function, is resolved in the binding environment of the closure's execution site on island J. It takes a predicate f as its argument, traverses the word database applying f to each word w, and returns a list.

Line 17 is the transmission by I to J of the request to evaluate the closure palindromes. The variable J@ is a CURL for island J denoting the target execution site for the palindromes closure. Finally, at line 18, the computation on island I waits (a maximum of 60 s, the lifespan of the promise) for the spawned computation to complete and return its result. If for some reason the spawned computation is unable to complete its task in the time allotted the result of the promise will be the value #f (false) given in line 18.

```
1  (let* ((sale
2          (resource/new
3           (resource/root)
4           3 (/ 1.0 17.0) (timespan/seconds 30 0 0 0)))
5         (day/even?
6          (lambda () (even? (date/day (date/now)))))
7         (path (list "books" "sale"))
8         (metadata
9          (list (cons "ISBNs"     (list b1 b2 b3))
10               (cons "gate"     day/even?)
11               (cons "discount" 0.80))))
12   (curl/new sale path metadata))
```

Fig. 2.4 Generating a time-limited, day-specific sales coupon as a CURL

The program of Fig. 2.3 is a classic example of moving computation close to the data that it demands and illustrates an effect that is difficult to achieve in a RESTful system; island J may easily host a large database of words, but it's not likely to implement a service expressly designed for extracting palindromes. However, that omission is irrelevant in COAST-based systems since a client is free to compose client-specific higher-order services from the primitives found in the execution sites of island J. No such provision exists in RESTful services.

Provider-Issued Mobile Code in CURLs A computation may embed closures as metadata in the CURLs that it issues and use those embedded closures as the interpreters of the messages that it receives. As the CURL is tamper-proof, the receiving computation (by definition the issuer of the CURL) may safely rely on any state and mobile code the CURL contains. When the computation first constructs and issues the CURL, it ensures that the CURL contains all of the static state (including arbitrary generated closures) that the computation will need in the future to serve the holder(s) of the CURL. In this manner computations, in addition to granting the capability to communicate, can enforce fine-grained constraints on the interpretation of messages. For example, a computation x may issue a CURL to y that allows y's mobile code, when sent to x, to call only one particular function that x selects and makes available.

For instance, an e-commerce site wants to issue CURLs as coupons for a book sale where three popular books, identified by ISBN numbers $b1, b2, b3$, will be on sale for a month at 80 % of the list price, but only on the even days of the month-long sale.

The construction of such a CURL is given in Fig. 2.4. Lines 1–4 specify the derivation (via function resource/new at line 2) of a CURL-specific resource account, sale, from the root account (line 3) of the computation. At lines 1–4 sale is granted a balance of three total uses, a rate limit of once every 17 s, and a total lifespan of 30 days (timespan/seconds at line 3 takes days, hours, minutes, and seconds and converts that span of time to total seconds). CURLs are intended for use by computations, not people; however, they can be serialized as text for storage in files or out of band transmissions such as email.

day/even? at lines 5–6 is a provider-generated MOTILE predicate that returns true if the current day of the calendar month is an even integer and false otherwise.

```
1  (define (random/new low high)
2    (let ((difference (- high low)))
3      (lambda () (+ low (* (random) difference)))))
4
5  (define (service/custom low high)
6    (let ((custom (resource/new (resource/root) 100 7.5 90.0))
7          (path (list "random" "custom" low high))
8          (metadata
9           (list (cons "implementation" (random/new low high)))))
10     (curl/new custom path metadata)))
```

Fig. 2.5 Generating a client-specific service as a CURL

date/now and date/day are provider-side calendrical functions. date/now returns the current date as a structure and date/day extracts the day of the month (1–31) from that structure. Line 7 defines the path for the CURL to be generated and lines 8–11 define the metadata to be included in the CURL as key/value pairs: the ISBNs of the books on sale, the gate function defined by the provider to determine the validity of the "coupon," and the amount of the sale discount. Finally, line 12 generates and returns the desired CURL.

When the e-commerce site receives a purchase request message sent by way of a "coupon" CURL it passes the CURL and message on to the book sale computation only if the message arrived prior to the expiration date of the CURL. The book sale computation executes the gate function contained within the metadata of the CURL to determine if the coupon is valid. If so it allows the purchase to proceed; otherwise the request is rejected. As the book sale computation is ignorant of the details of the gate function included in the CURL metadata the e-commerce site provider can easily generate customized sale coupons, each with different gate functions.

Service Implementations in CURLs Fig. 2.5 illustrates how a CURL can carry a service implementation; here, generating custom ranges of real random numbers. Lines 1-3 define a utility function random/new that returns a customized random number generator as a closure (line 3). The provider-side function random returns a real random number in the open range [0, 1]. service/custom returns a customized CURL for a client requiring a random number service using bounds, low and high, specified by the client. The CURL is granted a use count of 100, a rate limit of 7.5 Hz and a lifespan of 90 s (lines 6 and 10). The CURL metadata contains the custom random number generator (line 9). The CURL itself is the return value (line 10).

The server itself is just a skeleton that expects messages whose only content is a "reply to" CURL r. Recall that every MOTILE/island message is accompanied by the CURL u to which the message is directed. On receiving such a message the server extracts the service implementation (as a closure f) from the metadata of CURL u, evaluates f, and transmits that result via CURL r.

Non-Delegation and CURL Revocation COASTCAST [97] is a COAST-based service for the distribution and manipulation of real-time High Definition (HD) video. Islands whose assets include HD cameras and execution sites containing primitives

for managing cameras and encoding (compressing) video serve video streams to other islands with high-resolution monitors for displaying the video streams. Island assets may be less tangible, for example, islands with sufficient computing capacity and network bandwidth to relay high-bandwidth video streams to other less capable islands. In such applications camera islands may want to restrict direct access to cameras to a small set of trusted display or relay islands. In other words, if island I holds a CURL u granting it access to a particular camera of island J we would like to guarantee that only I may access the camera of J even if it hands CURL u on to island X for its use. This property, non-delegation, is enforced by embedding J-generated restrictions (as metadata) in the CURL u that J provides to I. As all islands are self-certifying, island J can determine authoritatively if a message m sent to it via CURL u was sent from island I or some other island X. Each CURL u may contain (as metadata) a predicate (a single-argument closure) that, given the address of an island, returns true if the island is permitted to use u and false otherwise.

Fig. 2.6 illustrates the construction of a CURL by island J that limits delegation on the basis of processor load. Islands A, B, and C are each permitted access to HD camera 3 of island J with a resolution of 720 p at a frame rate of 20 frames-per-second (indicated by the path, line 14). Combined, the three islands may access the camera a total of 10 times (line 1), at most twice per day (lines 2–3), over a period of 14 days (lines 4–5). Lines 6–7 define a derived resource account camera that enforces these use, rate, and lifespan constraints.

Lines 8–13 define the delegation predicate dictated by island J. The access of all three islands A, B, and C is determined by the current processor load on island J. A may access the camera only if the processor load is low (≤ 3.7), B may access the camera only if the processor load is moderate at worst (≤ 7.3), and C may access the camera only if the processor load is not excessively high (≤ 10.1). The CURL J@ for access to camera 3 is generated at line 16 and is distributed to islands A, B, and C at lines 18–20.

When a closure f is sent to the execution site of camera 3 of island J via CURL J@ island J applies the delegation predicate in the CURL metadata to the address of the transmitting island. If the predicate returns true then closure f is evaluated in the context of the execution site of camera 3; otherwise, closure f is rejected. Consequently, no other islands besides A, B, or C can access the camera and the access of these three is predicated on the current processor load. If another island X somehow acquires CURL J@ it cannot be used productively by X.

Any CURL issued by an island may be revoked at any time by that island. The unit of revocation is the resource account r on which the CURL draws; for example, the CURL J@ generated at line 16 of Fig. 2.6 draws upon the resource account camera constructed at lines 6–7. If the resource account r upon which a CURL u draws is invalidated by its issuing island then any message transmission via u will be summarily rejected from that point forward. If multiple, distinct CURLS $u_1, \ldots, u_n$ draw upon r then the invalidation of r revokes all such CURLs u_i.

```
(let* ((uses 10)
       (rate
        (/ 2.0 (timespan/seconds 1 0 0 0))) ; Twice per day.
       (lifespan
        (timespan/seconds 14 0 0 0)) ; Fourteen days.
       (camera
        (resource/new (resource/root) uses rate lifespan))
       (delegate
        (lambda (x)
          (or
           (and (eq? x A) (<= (cpu/load) 3.7))
           (and (eq? x B) (<= (cpu/load) 7.3))
           (and (eq? x C) (<= (cpu/load) 10.1)))))
       (path (list "camera" 3 "720p" 20))
       (metadata (list (cons "delegate" delegate)))
       (J@ (curl/new camera path metadata)))

  (curl/send A@ J@)
  (curl/send B@ J@)
  (curl/send C@ J@))
```

Fig. 2.6 Generate a CURL that limits delegation on the basis of processor load

2.5 Motile/Island: A Reference Infrastructure

The COAST style imposes substantive constraints on how COAST-based applications must be built. Satisfying these constraints with a typical imperative programming language is awkward so we have created an implementation platform for constructing and deploying COAST applications: MOTILE, a mobile code language whose semantics and implementation enforce key constraints on the use and migration of capability, and ISLAND, an infrastructure for MOTILE computations.

COAST Computations as Actors Each COAST computation is implemented as an actor [3]. Each actor is an independent thread of computation that may transmit asynchronous messages to other actors, receive asynchronous messages from other actors, conduct private computations, and spawn new actors. All four actions are implemented (and perhaps selectively restricted) by functions in binding environments. Spawning is implemented as a specialized kind of message sending. The assumptions of the actor model, private computation and asynchronous messaging, match those of COAST, where private computation is conducted only in the context of a specific execution site. Actors are distinct from agents as, unlike agents [36], each actor is immobile (closures and continuations are mobile but not actors). Also, in many agent systems the identity of the agent is invariant as it moves from host to host, whereas spawning a closure or continuation results in a new and distinct actor.

Motile MOTILE is a single-assignment, functional language for defining COAST computations. All MOTILE actors are named by one or more CURLs, a base data type in MOTILE. All MOTILE data structures are purely functional [173] (hence immutable). This choice reduces the semantic distinctions between messaging where sender and receiver share an address space and messaging where the sender and receiver occupy separate address spaces. Since all data structures (including messages) are immutable the data synchronization races common to shared-memory, imperative

languages are not possible. By implication, shared-memory attacks where values are mutated after being shared with other actors are impossible.

Island An *island* is a single, homogeneous address space occupied by one or more MOTILE actors. Islands implement the role of "execution host" discussed in Sect. 2.3. Each island is uniquely identified by a triple: the public key half of a public/private key pair, a DNS name, and an IP port number. All islands are self-certifying [155, 253] and all communication between islands is encrypted. Each island is instantiated with an initial set of execution engines, binding environments, and a set of trusted computations that allocate execution sites to visiting computations. Those trusted actors have access to implementation-level MOTILE primitives that other computations are not permitted to call; for example, creating an actor, instantiating island-wide user interfaces, and staging fixed island assets. These trusted actors also issue CURLs naming themselves, with the distinction that their CURLs are *durable*—valid even after an island is restarted. Computations holding a CURL for a trusted actor t are permitted to send a closure to t to spawn a new COAST computation. The specific execution engine and binding environment allocated to that new computation conform to the security and usage policy enforced by t.

Capability URL Implementation Every CURL u denotes a specific computation x and contains:

- An *address*, the public key, DNS name, and IP port number of island I
- A *path* (a list of MOTILE values, possibly empty), defining for x the domain of interpretation of a message sent via u
- The *resource key*, a globally unique cryptographic identifier [142], used by island I as an index to CURL-specific, island-side state (including CURL timestamps, use count, and rate limit)
- The *creation* and *expiration* timestamps of u. After the deadline (the expiration timestamp) any message sent via this CURL will be rejected
- A *use count*, a positive integer, giving the nominal maximum number of messages that may be delivered to x via u
- A *rate limit*, a positive number, giving the nominal maximum rate (in Hz) at which messages transmitted via u to x will be delivered to x
- Arbitrary metadata that may include primitive values, standard structures such as lists or vectors, other CURLS, closures, continuations, and binding environments
- A cryptographic signature (over the contents of u) generated by the island I on which computation x resides. The signing, based on the private key of island I, allows any computation holding CURL u to verify that u is a valid CURL for x on I

A CURL supports, by construction, four base restrictions:

- Use count (total number of messages per CURL)
- Expiration date (after which the CURL is invalid)
- Rate limits (rate of message transmissions per CURL)
- Revocation (permanently withdraw, per CURL, the capability to communicate).

All are enforced by the issuing island I, since no island would reasonably trust another to enforce its own restrictions, and all four restrictions require the issuing island to maintain a small amount of state. Let u be a CURL for actor x. A trusted actor of I inspects each CURL/message pair u/m on arrival, passing the pair onto actor x if and only if CURL u is valid and the pair satisfy all I-imposed restrictions. At CURL generation time, x and I may both insert arbitrary MOTILE expressions into u in addition to customizing the base restrictions listed above. In this manner x enforces x-specific, u-specific restrictions on communication including complex temporal constraints ("only on alternate Thursdays before noon"), use scenarios ("only legal expressions in a domain-specific language"), limits on delegation (only messages from island J) and conditionals based on observables ("the price of gold on NYMEX must be $<$ \$1657 per ounce"), and I in turn enforces restrictions it places on x and its collaborators. Each CURL contains the mobile code and static state that x will require to enforce those additional observable restrictions.

2.6 Related Work

COAST and MOTILE/ISLAND have been influenced by prior work on mobile code including remote evaluation [219–221], Scheme-based mobile code languages [89, 106, 245], the actor-like language Erlang [18], the object-capability language E [160], and capability-based operating systems [194, 208]. Island self-certification draws from self-certifying file systems [155] and URLs [115]. Our previous work on CREST [65, 66, 68, 96] inspired computation exchange and led us to consider the problem of secure decentralized services that COAST addresses.

The idiom of computation exchange is partially reflected in Emerald [114], a system devoted to high-performance object mobility. Like computation exchange, Emerald emphasizes fine-grain state and code transfers among hosts, but assumes a single sphere of authority, identical host processors, and extends no further than a local area network.

Kali Scheme [46] implemented distinct address spaces containing multiple threads (the equivalent of islands) as a language construction and introduced closure and continuation exchange in messages as a mechanism for spawning threads in remote address spaces.

Self-protective behavior for the sake of ensuring progress (liveness) and system integrity is a vital interest of local security. Resource sandboxing is a common defense mechanism to forestall denial of service attacks via resource exhaustion and is available in several languages including Java [145] and Racket [248]. Execution sandboxing denies executing programs unsafe access to critical resources. The Google Native Client [256] employs software fault isolation [246] to confine the execution of untrusted native Intel x86 code. Extensions to Native Client [16] adapt these techniques to the complex run-times of high-performance, dynamic, JIT-enhanced languages such as JavaScript.

Several mechanisms were employed by Telescript [148], an object-oriented, mobile agent system, for which security was a concern [225]. Mobile Telescript agents were executed by a host-independent virtual machine within *places*, virtual locations devoted to a particular service: for example, ticket purchases, or catalog search. Mobile agents and places were tagged with a designation of authority (the originating organization). Agents were granted *permits* by the managing authority of the place, which confined the capabilities granted to an agent and set resource caps. Telescript can be regarded as a mobile-code-based decentralized SOA.

Agent technology draws from both distributed systems and programming languages, notably for strong mobility. For example, Agent Tcl [131] (now D'Agents) had four principal goals: ease of agent migration, transparent communication among agents, support for multiple agent languages, and effective security. Agent Tcl implements "whole" agent mobility where the only unit of code mobility is the entire binary image of the agent and relies on Safe Tcl to confine the executing Tcl agents. A set of trusted Safe Tcl scripts provide limited access (based on access control lists) to unsafe functionality.

Object capability security is a pivotal influence on COAST. A capability [61], fuses access to, and designation of, a protected resource into a single, unforgeable reference. The object capability security model [160] implements confinement [208], revocation, and multilevel security [159]; offers patterns for non-delegation [163]; resolves the problem of the Confused Deputy [52, 107]; and is a base mechanism for information flow control [30, 162]. The Emerald language [193] is an early example of an object-capability language.

CURLs have precedent in the self-certifying URLs (YURLS) of Waterken [51], the unique URLs of Second Life[1], and the time-limited, signed URLs of Amazon S3[2]. YURLs embrace "communication by introduction" in which a client, interacting with a trusted partner, is granted the capability to communicate with a specialized service acting on behalf of (or equivalently, as a proxy for) the trusted partner. Both YURLs and CURLs contain one or more large, cryptographic numbers, in the former the SHA hash of the public key of a web site and in the latter, the public key of an island and the resource key of the resources account affiliated with the CURL. Consequently, both YURLs and CURLs are impossible to guess but YURLs can be forged as they are not signed. In contrast, since CURLs are signed with the private key of the issuer they cannot be forged, are tamper-proof, and are non-repudiable.

2.7 Conclusion and Future Work

Since decentralized services, by definition, have no single defensible perimeter, all of the constituent services must be self-defensive. Capability security is the principal defensive mechanism for COAST-based systems and takes two forms: functional

[1] http://wiki.secondlife.com/wiki/Protocol#Capabilities
[2] http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html

capability, circumscribed by the execution engine and binding environment of the individual execution sites of computations, and communications capability, where communication by introduction and Capability URLs limit and shape the ability of computations to inter-communicate. By design CURLs prevent arbitrary communication among service components and, by constraining communication, reduce the risks and consequences of both accidental errors and malicious attacks. Communication between computations x and y is possible only if at least one of the two holds a CURL for the other. However, that is the minimum necessary condition, since any messaging between the two must also satisfy a CURL-specific use cap (the total number of messages that may be sent), rate limit (the frequency in Hertz at which messages may be sent), and an expiration deadline (the "end of life" for the CURL). With these constraints a computation can regulate the total number of messages that it receives from another, the arrival rate of those messages, and the span of time over which it can expect to hear from another computation—all of which can thwart or reduce abuse of service and ensure fair service for others.

These basic constraints are useful but insufficient for enforcing service agreements based on *observables*; real-world phenomena (weather, processor load, stock prices . . .), the states of the communicating computations, or the states of computations elsewhere. CURLs, when combined with embedded MOTILE mobile code, facilitate: preconditions and use restrictions incorporating observables, explicit state transfer in the spirit of REST, service customization, service transfer for which both the state of the service and its implementation are completely explicit, and non-delegation that incorporates arbitrary temporal and use constraints. The combination of communication by introduction and mobile code is a significant contribution to the safety and security of decentralized services.

Mobile code embedded in CURLs can serve other functions as well, including logging, message tracing, debugging, exception handling, event distribution, traffic analysis, checkpointing, and service restart. Many interesting research questions remain; for example, domain-specific security languages or service-level contracts as embedded mobile code in CURLs, language constructions for incorporating, and responding to, resource restrictions in CURLs, hierarchical constraints in CURLs that reflect layered, system-level concerns, the roles of CURLs with embedded mobile code in dynamic software update, and COAST-like communication by introduction for embedded and soft real-time systems.

Acknowledgements We are indebted to Kyle Strasser whose implementation of COASTCAST broadened our understanding of communication capability in MOTILE/ISLAND and the means by which functional capability could be manipulated to support security. Our thanks as well to the anonymous reviewers for their incisive and helpful comments.

This work was supported by the United States National Science Foundation under Grants CCF-0917129 and CCF-0820222.

Chapter 3
Interoperability of Two RESTful Protocols: HTTP and CoAP

Myriam Leggieri and Michael Hausenblas

3.1 The Internet of Things (IoT)

The Internet as we know it, is about to change, becoming a global digital nervous system with consequences that deeply affect our lives. We have already entered a state of always on connectivity, where people interact not only with each other and common devices like PC and laptops, but also with things. Things interact with each other through machine-to-machine (M2M) communication, they have been on the Internet, but not directly connected to it or able to sense the surrounding real world. What is currently undergoing is a massive attempt at scaling up the amount of connected things on one side, and down the cost of connecting them on the other. It is expected that by using *the same* standards, the integration with applications will be simplified and higher-level interaction among resource-constrained devices—abstracting away heterogeneities—will be possible. In fact, devices embedded in any sort of objects are becoming natively IP-enabled and Internet-connected, while Internet services monitor and control them. The phenomenon of expansion from the original Internet set, is called the Internet of Things (IoT) (Fig. 3.1).

It starts from a core including a backbone of routers and servers with high capacity, rarely changing; goes through the Fringe—including devices with human assistance for connectivity e.g., laptop, PCs and, some mobile phones—and ends with IoT, where embedded devices communicate with each other directly or through the Internet without human intervention. IoT devices are key-enablers of enhancements in several applications, e.g. logistics, building automation, smart metering and industrial automation. This phenomenon has enormous proportions, e.g., the microcontroller and microprocessor market sales grow up to 43 billions USD in 2009; the amount

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

M. Leggieri (✉)
Digital Enterprise Research Institute (DERI), National University of Ireland,
Galway, IDA Business Park, Lower Dangan, Galway, Ireland
e-mail: myriam.leggieri@deri.org

M. Hausenblas
DERI, NUI Galway, IDA Business Park, Lower Dangan, Galway, Ireland
e-mail: michael.hausenblas@deri.org

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_3,

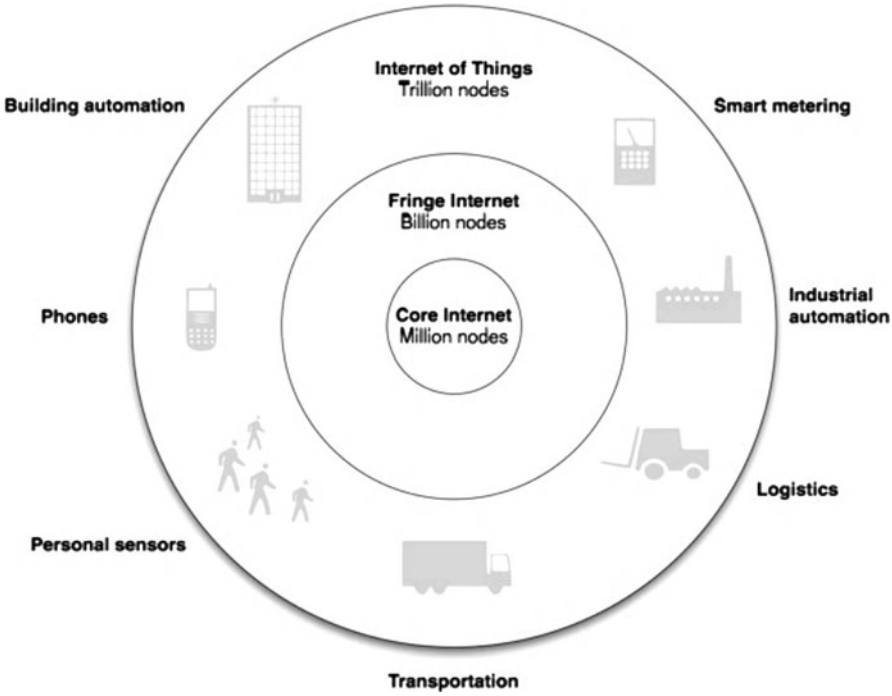


Fig. 3.1 The Internet of Things Vision: Expansion from the core Internet backbone

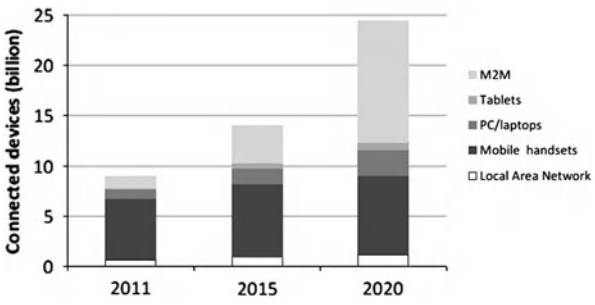


Fig. 3.2 Connected Life: exponential growth estimation of embedded devices by 2020

of Internet-connected devices (Fig. 3.2) is expected to grow [195] up to 50 billion USD by 2020 [72]. These estimates rise up to trillions if considering personal, local and wide-area networks.

Unlike the homogeneous PC Information Technology—mostly aimed at home and office environments—embedded devices span across disparate and money-saving applications, from personal health to large-scale facility monitoring, leading to the deployment of heterogeneous solutions. For instance, a facility management system automating energy consumption according to specific bill rates, would cause money and energy saving. However, realizing the IoT vision is not trivial, since IoT devices are characterized by resource constraints that make expensive to apply Internet protocols. IETF Working Groups are undergoing the definition of standard protocols

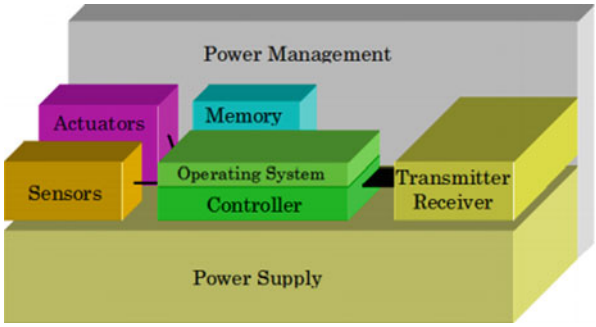


Fig. 3.3 Embedded Devices Architecture.

at different layers of the network stack to facilitate the translation into the Internet ones, i.e., 6LoWPAN [210]—IPv6, for resource constrained devices replacing the expensive fragmentation of IPv6 packets into small link-layer frames; and the Constrained Application Protocol (CoAP) [54], a downscaled version of HTTP on top of UDP for resource constrained machine-to-machine (M2M) applications. Specific protocol requirements for M2M are: built-in discovery, asynchronous message exchange, multicast support, low header overhead and parsing simplicity.

We describe the embedded devices we are referring to in Sect. 3.2, which leads us to define a set of requirements for resource-constrained environments in Sect. 3.3. For each requirement we analyze whether and how it could be satisfied by the current RESTful principles or new design directions should be required. We explain also how the main Internet, RESTful, and constrained environment's protocols, could currently deal with these requirements. We summarize the comparison these approaches in Sect. 3.4. In Sect. 3.5 we describe the guidelines for handling the increasingly common scenario of heterogeneous networks, where HTTP, CoAP and ad-hoc networks interact requiring a mapping between HTTP and CoAP. We conclude (Sect. 3.6) by depicting an IoT current state summary, predicting the next steps towards the full realization of the Future Internet vision.

3.2 Embedded Devices

Embedded devices are usually characterized by several resource constraints. Figure 3.3 illustrates their typical architecture, where power supply, management, memory (where the program code is stored), sensors, actuators, Input/Output and peripherals rely on a microcontroller, the main processing unit. Microcontrollers are special purpose and highly integrated with each of the components as in Fig. 3.3. The ratio between their power and performance is optimized and their low price (ranging between 0.25 and 10.00 USD) facilitates their dissemination among the average people.

The main constraints usually apply on power and memory. Embedded devices are often battery-powered, thus requiring them to enter into the sleep-mode for long period of time, up to months. Their memory has a limited size, often shared with

the program code and the operating system. They can be equipped with Random Access Memory (RAM)—included on-board in microcontrollers—Read-Only Memory (ROM), serial flash external memory and Flash erasable programmable memory, which can be read and written in blocks but with slow and power-consuming access. For instance, the ATMEL microcontroller has a limited RAM up to 4 kB and limited ROM up to 128 kB.

IoT encompasses a broad range of devices, with highly heterogeneous constraints and capabilities, which arise several network challenges. These devices are usually characterized by high loss and link variability (ranging around 100 kbit/s). The physical layer packet size may be limited up to 100 bytes. For instance, the IEEE 802.15.4, a popular low-power (1 mW) radio, ranges between 0.9 and 2.4 GHz bands at 868 MHz—according to the European directives—with 1 % duty cycle, 20 kbit/s and its packet size limited to 127 bytes.

Embedded devices can be classified as follows, in terms of their constraints, in order to deploy solutions that address the specific requirements of the class of interest. 1. Class 0 (C0) devices are the most constrained ones, thus rarely reconfigured or queried. They have to rely on proxies or gateways to participate in Internet communications; 2. Class 1 (C1) devices are able to host a lightweight protocol stack (e.g., CoAP over UDP) although not the Internet one; 3. Class 2 (C2) supports the Internet protocol stack, although they can also benefit from lightweight and energy-efficient protocols as the ones used by C1 devices. In this way, development costs are reduced and the interoperability is increased; 4. Power constrained devices are no constrained in terms of processor and memory, but have a limited energy supply.

3.3 Architectural Design for Constrained Environment

Architectural decisions have to be made, leveraging the specific embedded devices features, described in Sect. 3.2. Our contribution to the architectural design process, consists in discussing on strengths and weaknesses of the current RESTful principles and main implementations, in a resource constrained context. We achieve this, by describing the specific requirements [147] characterizing such environment (Fig. 3.4), how much they could be covered by the REST architectural principles and by which technical (Internet or CoRE) protocol implementation. Typical requirements are asynchronous transactions, service and resource discovery, reliability, multicast and simplicity. In general, traditional Internet protocols present several limitations that make them not suitable for constrained environments, mostly because of their overall complexity.

Figure 3.5 compares the typical IoT protocol stack with the traditional Internet one. The choice of protocols to consider, is based on their widespread deployments. At the application layer, we refer to HTTP on the Internet side and to CoAP for the IoT. The *Constrained Application Protocol* (CoAP) [85] is the main focus of the IETF's Constrained RESTful Environment Working Group (CoRE WG), aimed at enabling resource-oriented applications on constrained networks, fully compliant

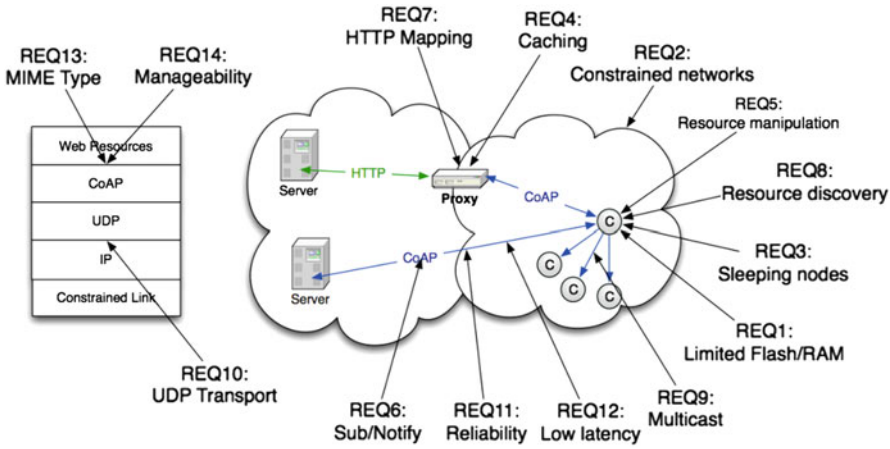


Fig. 3.4 Typical constrained environment architecture and its requirements

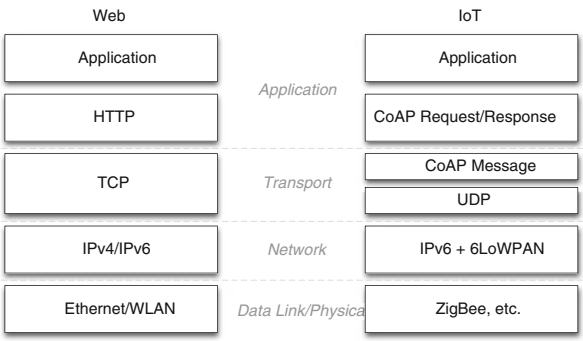


Fig. 3.5 Correspondence between the protocol stacks of the Web and the IoT

with the RESTful architectural principles. CoAP is a lightweight, binary protocol that supports a subset of MIME types and HTTP-compatible method codes.

At the network layer, we refer to IPv6 on the Internet side and to 6LoWPAN for the IoT. We believe that the transition to IPv6 is ineluctable, since the current 4 billion IPv4 addresses are almost exhausted, and the number of Internet-connected devices is growing. Nonetheless, IPv4 interconnectivity is easily achievable if necessary, e.g., by applying IPv6-in-IPv4 tunneling and address translation. On the IoT side, the IETF's IPv6 over Low power WPAN Working Group (6LoWPAN WG) has recently finalized its activity, defining 6LoWPAN to enable the efficient use of IPv6 over constrained devices in constrained (low-power, low-rate) wireless networks, by relying on an adaptation layer and protocol optimizations.

3.3.1 Reliability and Resiliency to Heterogeneity

The management architecture should support devices from any of the classes in Sect. 3.2, but also scale with any network size and topology. It should be distributed, to overcome the lacks of server connectivity by providing higher reliability, while dealing with the cost of the increased complexity. Then, the management protocol should be extensible to support changing requirements, to scale and provide a high degree of resilience to support loose and unreliable links, high transmission error rate, limited data rate and high latency.

To achieve this scalability, in addition to caching and aggregation techniques, hierarchical management models can be deployed (top-down network configuration), i.e., providing intermediary entities that take the responsibility for managing subsets of the constrained network devices. REST principles with respect to dealing with heterogeneity issues, have already been implemented in the Internet, to enable web services cooperation and web applications. However, the HTTP approach does not suit, since the HTTP clients support for the Expect header is poorly implemented, i.e., clients often do not wait for the 100 Continue response from the server, before beginning to send their requests. This could overload resource constrained devices. At the Transport layer, TCP does not support the requirement for reliability, since it can not detect congestions by distinguishing between packets dropped or lost on wireless links. IPv6 offers almost unlimited scalability—e.g., by its address space of 2128 addresses instead of the 232 IPv4 ones—thus enabling peer-to-peer connectivity and solving the NAT barrier—with specific and permanent IP addresses—for any Internet-connected thing. This also grants seamless connection while moving from one Internet access point to the other, in mobile scenarios. IPv6 supports multicast and any kind of cast functionalities, self-configuration mechanisms, security and authentication features, thanks to the mandatory IPSec capacities and the possibility to use the address space to include encryption keys.

Reliability is enabled by the CoAP definition of specific types of messages and of the procedure to process them. A CoAP message can be classified as 1. Confirmable (CON) or Not-Confirmable (NON), in case the sender is going to wait to receive a confirmation of successful reception from the recipient, or not; 2. Acknowledge (ACK), as a successful reception of a message, sent by the recipient, so that the client will stop retransmitting the request. This is called *piggy-backed response* (Fig. 3.6).

If the received message is a request that the server is able to satisfy but not immediately, then the ACK message will be empty and followed by another CON message containing the answer from the server, as soon as possible. While all the exchanged messages will have the same Message ID, this last CON one will have a different Message ID but it will still be possible to associate it with the correct previous interaction by the *Token option* whose ID, indeed, will be the same all over this specific exchange. This is called *separate response* (Fig. 3.7).

Reset (RST), in case the recipient is not able to either process a request, or provide a suitable error response or process it later. This is not compulsory and would be returned instead of an ACK message. One of these message types is specified in

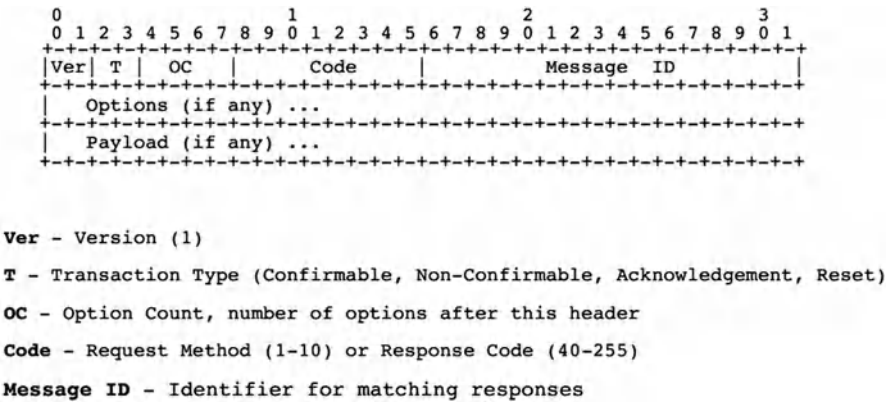


Fig. 3.6 Piggy-backed response example

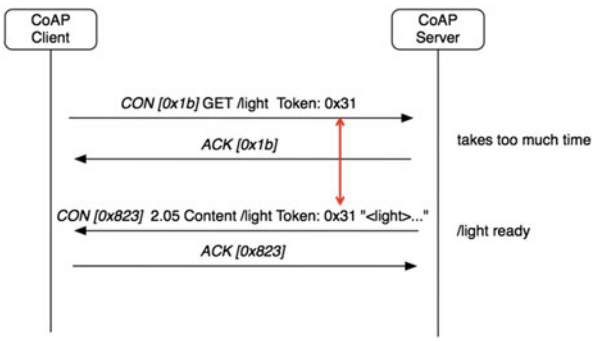


Fig. 3.7 Separate response example

the T field of the CoAP header. The Confirmable message enables reliability. They are never empty and are retransmitted at an exponentially increasing rate, until they either reach a timeout or get acknowledged with an ACK (empty or not) or an RST, which all have to echo the same Message ID as the CON's one. CoAP relies on the client's responsibility to keep strictly limited the amount of outstanding connections, (i.e., CON requests or other kind of requests for which an ACK or a response has not been received yet) with the same server. However, also servers should apply a rate for its response transmission, according to the specific application requirements, for congestion controlling in case of client malfunctions or attacks.

RESTful principles easily scale while do not specifically address reliability. A related issue is the necessity for continuous connectivity which is covered by 6LoWPAN in the IoT stack. A LoWPAN is a collection of 6LoWPAN Nodes which share the first 64 bits of their IPv6 address. Consequently, as long as a node moves inside a LoWPAN, its IPv6 address never changes (respectively, it changes as the node moves to another LoWPAN). The 6LoWPAN overview architecture is presented in Fig. 3.8. It consists of nodes as hosts or routers, that can belong to one or more LoWPANs

at the same time (multi-homing). LoWPANs can be classified as 1. Simple LoWPANs, i.e., connected to another IP network through an Edge Router; 2. Extended LoWPANs, i.e., backbone link interconnecting multiple IP networks by their Edge Routers; 3. Ad-hoc LoWPANs, i.e., LoWPAN that operates offline.

Edge Routers are responsible for routing traffic both inside and outside the LoWPANs, handling compression and transparent, stateless bi-directional adaptation between full IPv6 and LoWPAN format. This is required only outside of the LoWPAN. These routers are also in charge of managing the Neighbor Discovery (ND) algorithm process, where nodes register their presence to an Edge Router; and are Internet-connected through backhaul links, e.g., GPS, DSL (as in Fig. 3.8). The Simple LoWPAN and Extended LoWPAN Nodes communicate in an end-to-end manner, are identified by permanent and unique IPv6 addresses and are able to send and receive IPv6 packets. Since often the LoWPAN adaptation layer (deemed to optimize IPv6 over IEEE 802.15.4 and similar link layers) is implemented together with IPv6, they can alternatively be shown together as part of the network layer.

3.3.2 Asynchronous Reliable Messaging

In constrained environments, documents are not large, thus not requiring document range and response continuation; and they are not meant for direct user consumption, thus not requiring language and charset negotiation. Mainly the interaction model needs to be asynchronous as a consequence of the resilience requirement detailed in Sect. 3.3.1, in order to avoid high transmission error rate and unreliability to cause network disruptions. In the integration between constrained and non-constrained networks, lossless automated mapping between management protocols should be possible, and the data models used in each network should be consistent with each other and automatically mappable. Defining an underlying information model design is a key-enabler for future model reuse and interoperability, thus constitutes a strong requirement. At least one management interface, should collect and expose information about the device status, energy parameters, usage, available resources and their estimated remaining availability.

Because of duty-cycling and of resource constraints, embedded devices can not be considered able to support synchronous interactions as traditional REST implementations do. The roles that are usually associated with a RESTful server are reactiveness, storage of authoritative versions of resources and management of namespaces and resource states. Web services rely on TCP, HTTP, SOAP and XML, as depicted in Fig. 3.9, which require complex transaction patterns. Also the Simple Network Management Protocol (SNMP) is usually inefficient and complex.

IoT devices can enter the architecture as either clients or servers or both. CoAP relies on an interaction model that is similar to the client/server one of HTTP, since requests for an action (associated with a method code) are sent by a client on a resource identified by a URI, and hosted by a server. However, for M2M,

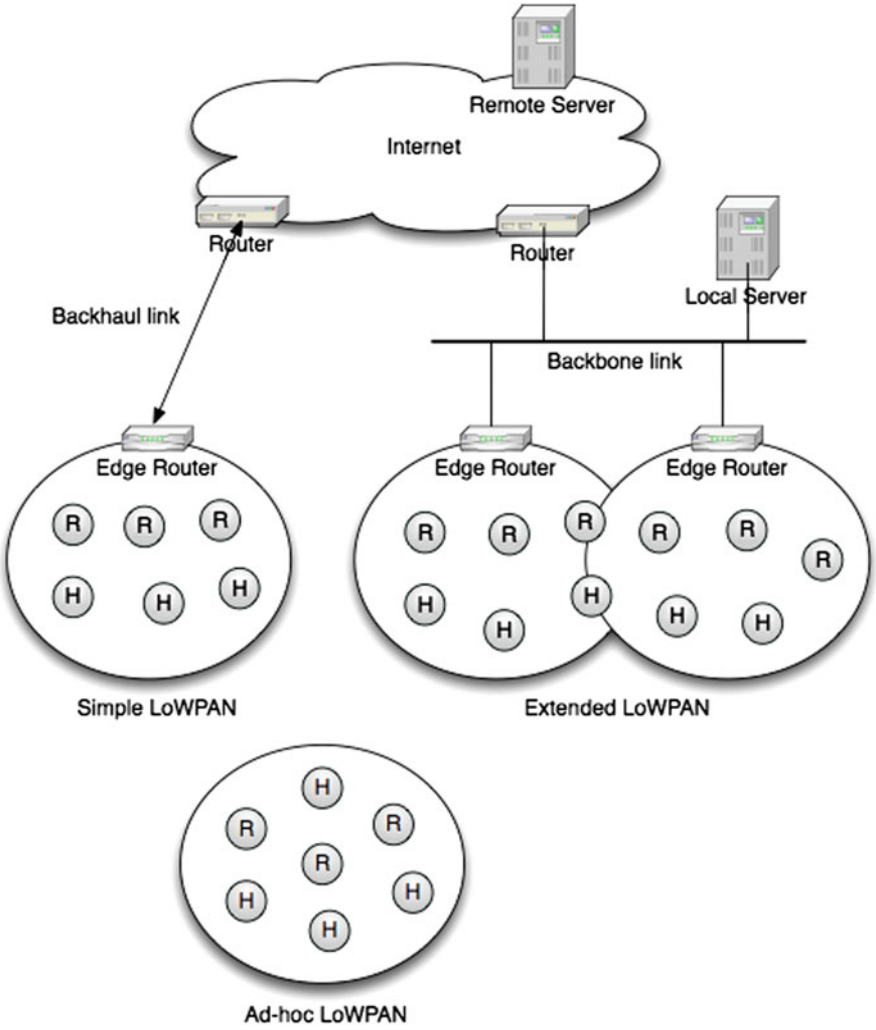


Fig. 3.8 6LoWPAN Architecture

the request/response interaction happens asynchronously over a datagram-oriented transport layer, i.e., UDP. This is supported by CoAP through messaging, as depicted in Fig. 3.5. Constrained devices tend to fit in the server role, providing sensor readings and capabilities as resources to client, as in Fig. 3.10. As servers, they benefit of continuity, eventually enabling on-demand merging of data, and of scaling—since clients might not be stateless and, then, might not scale. Also, constrained devices when considered servers exposing readings and metadata as resources, can leverage on the extensibility of the resource model.

The interaction model asynchronicity is one of the most inherent characteristics—thus most common—of constrained environments. Then, although even following

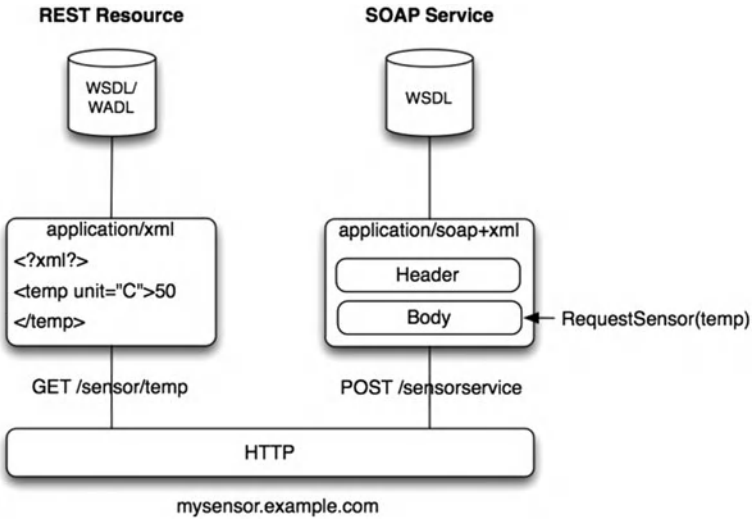


Fig. 3.9 Web Service Paradigm

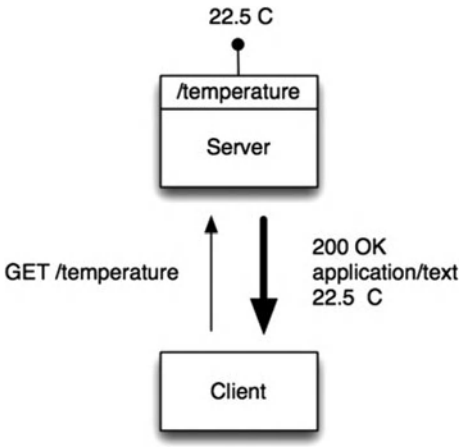


Fig. 3.10 REST request example

RESTful principles, asynchronous messaging can be simulated—by defining a *queue* resource representation on which the RESTful actions apply—a specific design for this setting should be defined.

3.3.3 Data Representation Modeling

To configure networks in a top-down manner—as requested in Sect. 3.3.1—the configuration data about individual devices should be abstracted in a network-wide setting. At the same time, the resource-constraints require the management data to be compact and space efficient, enabling small message sizes to save memory. In

general, complexity—e.g., application layer transactions that require either large application layer messages or the message reassembly at multiple layers in the protocol stack—should be avoided.

Applying data compression or data encoding techniques, can help achieving this result. However, compression requires additional code size, buffer and state information; thus it is usually feasible only in mobile application scenarios, to reduce transmission time and bandwidth, as long as C0 devices are not involved.

RESTful resources are abstract entities, whose representation is provided to clients as hypermedia, i.e. information and controls through which the client can select choices and actions. Hypermedia includes temporal anchors within a media stream; which in our IoT scenario could translate in anchors for fastening the access to sensor data streams. The resource representation can be provided in any format, which can be negotiated; while the access is granted through a uniform interface. However, the HTTP implementation usually adds complexity independently from the chosen resource representation, because of the protocol specification itself. On the complexity side, parsing headers with quoting, separators—e.g., comma and semicolon—whitespace and continuation rules is expensive. Also, the HTTP Header is carried along HTTP responses with no specific requirements for encoding and parsing efficiency. For this reason, there is no specific support for defining the maximum resource size, either; while this would be useful for embedded devices. Several different ways to indicate the content-length are allowed, besides the HTTP specifications, and require support by both clients and servers.

At the network layer, IPv6 requires support for multicast, which is extremely resource consuming and not available for radio technologies (e.g., IEEE 802.15.4). IPv6 messages also need to be fragmented since they use more bytes than the average amount allowed to be transmitted in constrained networks. For instance, a minimum of 1280 bytes, while the IEEE 802.15.4 standard has a 127 bytes of frame size boundary limit, with the layer-2 payload size as low as 72 bytes.

CoAP is better designed to avoid complexity. CoAP options and the URI scheme are fundamental for getting the most compact representation possible, as required in environments that are usually constrained for bandwidth and energy. CoAP messages are exchanged over UDP between endpoints and are identified by transport layer multiplexing information, e.g., UDP port number and security association. Each message contains a 4 bytes length binary header, a *Message ID*—to detect duplicates and to support reliability (Sect. 3.3.1)—eventual binary CoAP options and a payload. A CoAP message also carries request and response semantics as Method code or Response code; while other metadata—URI and payload media type—are part of the CoAP options section. CoAP options are never mandatory but some of them are classified as *critical* since, when not supported, must cause any kind of request to be rejected. Orthogonally to this distinction, CoAP options can also be marked as *safe* or *unsafe*. If not supported, *unsafe* options must always cause any request to be rejected while *safe* ones should always be forwarded by the proxy to the server hosting the requested resource (*origin server*). The Code field of the CoAP header, specifies whether a message contains either a request, a response or it is empty, according to the definitions in the CoAP Code Registry [85]. To further support a

compact data representation and transmission, the ETag option allows to indicate the specific version of a resource that is preferred, among the ones that vary over time. This also allows making requests conditional on the existence of a more recent version of a target resource, when used in combination with the If-Match option (or, as well, requests conditional on its non-existence, when combined with the If-None-Match option). This lowers the client/server communication, while RESTful format negotiations usually require more.

Also, when a CoAP server is hosted by a 6LoWPAN node that supports a port number in the compressed UDP port space, header compression efficiency can be improved. The LoWPAN format can be used to compress UDP, compacting down the header to 6 bytes, as shown in Fig. 3.11. TCP, because of its inherent complexity and inefficiency, could not be used in combination with 6LoWPAN.

The necessity to split the resource references is a direct consequence of the requirement for a compact data representation. URIs have to be shorter for resource efficiency, eventually decomposed in relative references. Resolving relative URIs using HTTP can be complex. Also, despite the HTTP specification directives, this kind of URIs are often found in the Location header where absolute URIs are expected.

For energy efficiency reasons, the normal IoT approach is not to encode URIs (still encoded or decoded URIs are considered equivalent) and to make use of specific CoAP options in order to decompose them, in a way that the full URI can be reconstructed at any involved endpoint. The host, port, path and query components, are removed from the URI and added as values of, respectively, the Uri-Host, Uri-Port, Uri-Path (one for each portion of the path, excluding the slashes) and Uri-Query (one for each parameter, excluding the question mark and the ampersands) options. Similarly, the Location-Path and Location-Query options specify, respectively, one segment of the absolute path to the requested resource and the argument parameterizing it. When combined together, they indicate a relative URI consisting of an absolute path, a query string or both.

Avoiding complexity in the URI scheme is equally important, since URIs have to be transmitted as well, and processed, in constrained environments. Figure 3.12 illustrates the COAP's URI scheme. The host component can be an IP literal, IPv4 address or a registered name—in this case requiring a name resolution service, e.g., DNS -. It is followed by the UDP port specifications, which defaults to 5683. Similarly to the HTTP URI scheme, resources are organized hierarchically as resembled in the path component, which is followed by a query one, where a sequence of arguments (usually in a key-value form and separated by ampersands) further parameterizes the requested resource. Short but still descriptive URIs are encouraged.

As a conclusion, though CoAP follows the REST principles, it does not focus on the concept of abstraction between a resource and its representation. While this is justified by the more urgent necessity to compact the protocol itself, it should not be overlooked since it could also support the data transmission reduction, e.g., avoiding to send again data that refer to the same resource as an already transmitted one, but in a different representation. Also, in the case of sensor data, a CoAP option to specify their space and time for identifying specific data instances, could be more appropriate than identifying them by a generic *version* concept (via the CoAP ETag option).

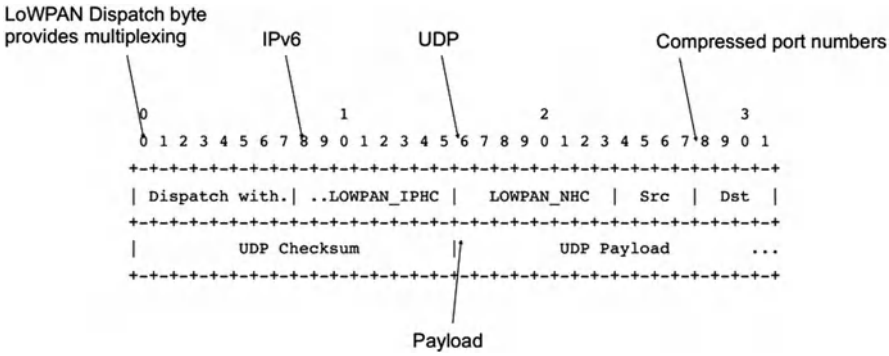


Fig. 3.11 6LoWPAN-UDP Headers which compact down to 6 bytes

```
coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
```

Fig. 3.12 CoAP URI scheme

3.3.4 Loose Coupling

Constrained devices often duty cycle themselves to save energy, intermediaries need to be set in the network, to provide information on behalf of sleeping nodes or support them in re-synchronizing after they wake up. Automatic synchronization is relevant in order to keep consistency, although the network must tolerate temporary inconsistency by properly distributing configuration parameters. Recovering from inactivity or self-reconfiguration are activities that might imply restarting devices. Then, the amount of state on each device must be minimized. Self-configuration should be enabled also in case of failures and during the initialization phase, despite additional information about the network topology might be unknown or uncertain; eventually combined with self-monitoring, to quickly detect faults and recovery (event-driven self-healing). It should be possible to apply delay schemes to incoming and outgoing links on an overloaded intermediary node.

According to the REST principles, the application state is moved to the client-side, while the resource state relies on the server-side. RESTful resources contain both data and hyperlinks that represent valid state transitions. Clients keep a correct application state only by navigating through hyperlinks. The client state then, affects the resources access, rather than the resources themselves. However, this raises security issues, since clients are allowed to provide a false client state to the server. The state is exposed as a resource itself, rather than being hidden in a session, thus enabling reuse, too.

A RESTful resource identification must be independent from the interaction. As a consequence, a URI should allow any scheme, avoid to include either any description of methods or fixed resource names or hierarchies (to not couple client and server) and should be used for defining extended relation names and hypertext-enabled markup for existing media types. In this way, servers keep complete control over their own

namespace, while separating instructions for clients on how to build correct URIs defined in media types and link relationships. HTTP stores the client state on Cookies, which constitute a more convenient approach than embedding the state in every resource representation. However, this approach causes side-effects, since the client-side state is domain-specific—rather than embedded in the resource representation - and is expensive.

The resource state on a CoAP server can change over time and it is relevant to notify the interested clients, of these changes. An extension to the CoAP protocol called *CoAP Observe* has been proposed [108], based on the observer design pattern [90] where observers register at a specific provider (the subject) to be notified whenever its status changes. Figure 3.13 shows the application of this pattern in CoAP, so that subjects corresponds to resources in the namespace of a CoAP server, and observers correspond to CoAP clients. The registration requires sending an extended GET request to the server for both being added to the list of observers of that specific resource and getting notified of the state updates. Figure 3.13 depicts a CoAP client that receives a notification with the current state upon registration and then two notifications as the resource status changes. Note the presence of the Observe option and the echo of the token, specified in the original client request that easily correlates the request with the future notifications. The server removes a client from the list of observers as soon as an ACK message is skipped, in response to a notification. This could also be due to packet loss. In this case, the client registers again, as soon as the last notification's expiration date is over and no updates have been received yet. This is compliant with the REST principles, since the server is responsible for the state and representation of resources in its namespace, while the client is responsible for keeping the application state, and the stateless exchange of resource representations. Intermediaries multiplex the interest of multiple clients in the same resource into a single association, for efficiency and scalability. The Observe protocol extension also grants consistency, since all the registered observers have a current representation of the last resource state. In order to support clients subscription to changes on only part of a resource representation status, the CoAP option Condition has been introduced [144]. It allows the client to specify which condition must be verified for the notification response to be sent. This procedure is called *Conditional Observe*.

We can conclude that the request for loose coupling is well covered by the RESTful principles and, especially, by the CoAP implementation. However, keeping the application state by just following hyperlinks is inefficient in constrained environment, as we explain in Sect. 3.3.5.

3.3.5 Driving the Application State for Resource Discovery

Resource discovery is the process where a client queries a server to get a list of hosted resources. A best-effort multicast can ease such discovery. This is extremely important in M2M applications because, on one side, static interfaces result in fragility, since there are no humans in the loop. On the other side, the management protocol

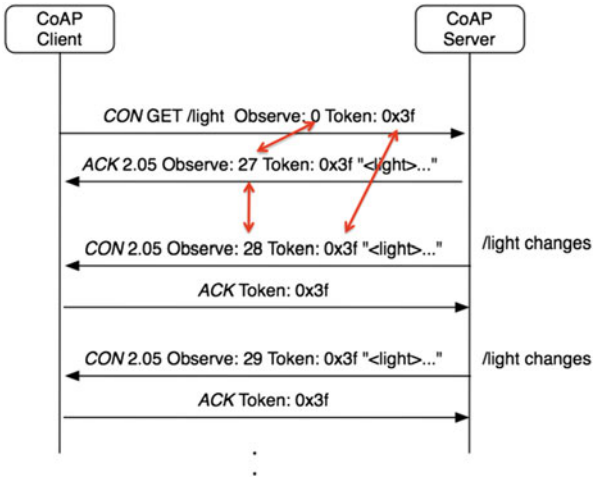


Fig. 3.13 Observing a resource in CoAP. Observer Design Pattern implementation

has to allow implementations where only a basic set of its primitives are supported, to ease its deployment on highly constrained devices; thus it should also be possible to discover which primitives (and which optional management capabilities) are available on each device. Also the network topology capabilities should be discoverable and monitored, e.g., by using a topology extraction algorithm. The resource discovery scenario can be split in the resource-directory and resource-collection sub-scenarios. The first one, typically involves sleeping servers or bandwidth limited access in constrained networks, where it is desirable to provide a directory of resources stored on other servers, without requiring access to each of them. The second one involves common data collection nodes, e.g., to get a list of faults in an issue tracker.

RESTful resources and their states are retrieved, accessed and used by navigating links, i.e., the resource representation is provided as hypermedia (see Sect. 3.3.3). Clients should access the resources provided by a REST API, with no prior knowledge other than the initial URI and the set of standard media types. Afterwards, the client could select the server-provided choices to drive the application state transitions according to its local context. Automated agents base their choice on the knowledge about typed relations, specific media types (every standard media type is associated with a default processing model) and action elements that they share with the server, i.e., a shared vocabulary which is exposed in the resource representation and which the agents should be adaptable to.

A fundamental difference between the Web and sensor network realms relies on the importance of the relation between a server and its hosted resources. While the HTTP negotiation between a browser and different web servers to get the requested content, is transparent and less relevant to the user, in constrained environments the main discovery use case consists in getting acknowledged of which resources are hosted by which server. On the Web side, The HTTP Link Header Field, aimed at enabling the discovery of further information about a resource. However, headers defined by the HTTP specifications other than [80] use incorrect syntax rules; while

the standard OPTIONS request that should allow to discover the server capabilities is poorly implemented. Links that grant access to further information associated with a resource, constitute the fundamental structure of the Web but are an exception in CoRE, where links are most aimed at the discovery of the resource itself, in machine-to-machine (M2M) applications. For this reason, carrying link information about a resource along with the protocol response, as defined by HTTP, should be avoided in CoAP and constrained environments in general.

The CoRE WG has defined the CoRE Link Format (CoRE-LF), i.e., a particular serialization of typed links which extends the format of the HTTP Link Header field serialization [209]. It is carried as a resource representation of a well-known URI, rather than along with the protocol response. In addition, it does not require special parsing (being encoded as UTF-8, it can be compared bit-wise). In order to support the described multiple resource discovery scenarios, the CoRE Link Format specification defines: 1. the relation type host to link a resource with its hosting server; 2. attributes for describing link metadata, which are detailed in Sect. 3.5; 3. resource registration to a resource directory by forwarding POST requests to an entry point. The resource directory lists resources as links in the CoRE Link Format; 4. resource discovery on a resource directory by forwarding requests to a resource directory lookup interface; 5. resource collections discovery by forwarding requests to an entry point. The resource collection lists resources either as links in the CoRE Link Format.

The entry points are referred to, using the path prefix /.well-known/core (Fig. 3.14) which is considered a well known location in the namespace of a host and used to enable the discovery of the host's metadata. To locate each entry point, a unicast resource discovery can be performed whenever a server's IP address is already known; otherwise, a GET request to the appropriate multicast address can be forwarded, assuming that the scope in which to search for a resource is limited, IP multicast is supported and precautions are followed to avoid response congestion.

The example in Fig. 3.14, shows a client discovering all the links to the resources hosted by the queried CoAP server, and their metadata; by addressing the well-known endpoint and interpreting the CoRE Link Format information. In this case, also the content type supported for each resource is also specified.

CoRE-LF defines the following attributes to specify link metadata, with a focus on identifying the type of resource that the link refers to, how to interact with it and evaluating the cost of addressing a request to it: 1. Resource Type (rt), i.e., which is a classification of the linked resource. It is meant to get a picture of the resource capabilities at a glance. For instance, rt="OutdoorHumidity", indicates the type of resource and, *implicitly*, the observed property (humidity) and its location. The relation type rt="host" is the default one; since, as explained, it is of interest in the most common resource discovery scenarios; 2. Interface Description (if), i.e., description of the exposed REST interface, either by referencing a WADL file or ad-hoc identifiers; 3. Maximum Size (sz), i.e., maximum size of the object returned when dereferencing the linked resource. This is relevant to determine whether requesting a resource would imply exceeding the Maximum Transmission Unit (MTU).

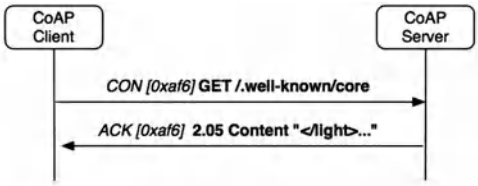


Fig. 3.14 Resource discovery example

```
</light>;rt="Illuminance";ct=0,
</s/maastr.xml>;title="Maastricht weather";ct=1,
</s/maastr/temp>;title="Temperature in Maastrich";ct=1,
</s/oulu.xml>;title="Oulu weather";ct=1,
</s/oulu/temp>;title="Temperature in Oulu";ct=1,
</s/temp>;rt="Temperature";ct=0
```

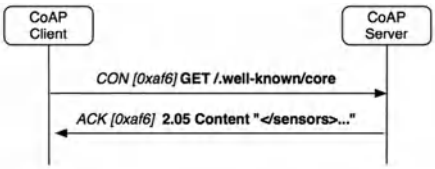


Fig. 3.15 CoAP messaging between client and server

```
REQ: GET /.well-known/core

RES: 2.05 Content
</sensors>;ct=40;rt="index";rt="Sensor Index",
</sensors/temp>;rt="TemperatureC";if="sensor",
</sensors/light>;ct=41;rt="LightLux";if="sensor",
<http://www.example.com/sensors/t123>;anchor="/sensors/temp"
;rel="describedby",
</t>;anchor="/sensors/temp";rel="alternate"
```

In the example in Fig. 3.15, during a piggy-backed request/response interaction, the CoAP server returns a payload in the CoRE Link format, where the resource type and interface description fields are used. In this case all the linked resources described, expose the same REST interface, i.e., if=sensor, although the resource types are all different, i.e., rt=LightLux and rt=TemperatureC. In particular, these resource type names, implicitly indicate also the units of measurement-i.e., lux and Celsius by appending the unit symbols as suffixes.

The Link Format is also meant to support situations in which a resource is too constrained to answer most of the direct requests. In this case, rather than referencing it, the link would consist of only a description of the relation type, while a URI reference to the hosting server can be contained in the CoAP attribute "Context". However, to properly interact with a CoAP-enabled device, there are some dynamic parameters that is necessary to know in advance, i.e., the supported CoAP options (especially if critical ones rather than optional) and content formats.

These parameters can be specified by the CoAP Profile Description Format. A profile about a resource can be retrieved by accessing the entry point /.well-known/profile?path=<resourceID>, where the profile fields op and cf are used to list

the CoAP options and the content formats that the resource hosting server supports. When block-wise transfer is used, the profile fields b1s and b2s allow specifying which block sizes are supported for Block1 and Block2. The example in Fig. 3.16, shows a list of profiles for two different resources, i.e., resourceID1 and resourceID2, and a default profile that applies to all the remaining resources hosted by the server. This list can be filtered by using the "Uri-Query" CoAP option, e.g., the value "cf=50" allows to retrieve only the profile of those resources for which an *application/json* format is supported, thus filtering out the default profile in the Fig. 3.16.

The RESTful use of hypermedia in M2M applications involves interacting with automated agents, as contrasted to the user deciding which HTML link to follow in order to interact with a Web server. There should be then, server-controlled hypermedia to advertise and navigate the server's resources. While the RESTful architectural principles, make design resilient to protocol and vocabulary changes, it causes two issues in constrained sensor networks. First, it leads toward too verbose requests exchange between server and client, for navigating resources until the one of interest is found and the correct content format is negotiated. Second, in time-critical scenarios, the resource representation might have changed during the range between the time it had been fetched and the time the next action had been chosen; thus becoming invalid.

3.3.6 Secured Communication

The transport protocol should be scalable, reliable and secure, providing authentication, data integrity, and confidentiality. Security is particularly important for constrained environments, since they are manipulating highly personal data as the ones collected from the real, personal and everyday people's life. The data should be granted by enabling access control on managed constrained devices and management systems, supporting security bootstrapping mechanisms, space and time efficient cryptographic algorithms.

The RESTful approach to security consists in point-to-point secure communications, using integrity checking and encryption. However, the HTTP protocol implementation is too verbose, while the inter-device communication should be reduced as much as possible. IPv6 includes optional support for IP Security through IPsec authentication and encryption, but the web services techniques to deal with this—like secure sockets or transport layer security mechanism—are too complex. Also, network limitations and socket-based security, might prevent the use of the full IPsec suite, enabled by IPv6.

The CoAP's URI scheme addresses the necessity to secure UDP datagrams through the use of DTLS, for privacy reasons. CoAP requests are never public and the resources that they address, never share the identity to those addressed by CoAP requests (which also applies when the same host on the same UDP port is invoked). However, CoAP is subject to the security leaks inherited by UDP, (although several approaches are applied to narrow the leak consequences). For instance, CoAP defines a meaning for the entire range of encodable values; reduces complexity caused

```
Request: GET coap://www.example.org/.well-known/profile

Response: 2.05 Content (application/json)
{
"profile":[
        {
                "path":".",
                "op":[11],
                "cf":[0],
                "b2s":[4,5]
        },
        {
                "path":"resourceID1",
                "op":[4,11,35],
                "cf":[0,50],
                "b2s":[4,5]
        },
        {
                "path":"resourceID2",
                "op":[4,35],
                "cf":[50]
        }

]
}
```

Fig. 3.16 CoAP request for the profiles of all the resources stored on the coap://www.example.org/ server, i.e., resourceID1, resourceID2 and others associated with a default profile description. The supported CoAP options, block sizes and content formats are mapped with numerical IDs, used in this example as values for the op, cf and b2s profile fields

by redundant representations of the same resource or by parsing, since it moves the URI processing responsibility to the clients.

Proxies constitute another source of vulnerabilities. Since they are allowed to break any IPsec or DTLS protection that a direct CoAP message exchange might have, they usually are subject to attacks for breaking availability, confidentiality or integrity of CoAP message exchanges. Finally, applications should try to minimize the packet and payload sizes, so that the performance penalty—due to the message fragmentation and reassembly applied by 6LoWPAN—can be reduced. It is important to comply with the 1280 byte Maximum Transmission Unit (MTU) size required by IPv6.

Table 3.1 Coverage of constrained environments' requirements by the RESTful principles, the IoT protocols and the Internet protocols, respectively

Design vs Requirements	H	CC	R	ND	AI	IMI	Simp	LC	RD	SC
REST	V	V	S	S	S	V	V	V	S	S
IoT protocols	V	V	V	V	V	V	V	V	V	S
Internet protocols	V	V	S	S	X	S	X	S	S	S

3.4 Discussion

To summarize, the REST architecture has proven to be robust enough to cover almost any of the constrained environments' requirements, as shown in Table 3.1. However, some of them should be explicitly addressed - because of their critical importance - while they are currently only inefficiently covered by the RESTful principles, adding complexity to the system. In Table 3.1, these requirements, which are currently covered but inefficiently, are indicated by *S*; while those that are completely covered are indicated by *V* and those that are not covered at all are indicated by *X*. The requirements' names have been shortened for convenience. They are, respectively, (1) Dealing with Heterogeneity (H) (2) Continuous Connectivity (CC) (3) Reliability (R) (4) Neighbor Discovery (ND) (5) Asynchronous Interaction (AI) (6) Information Model Interoperability (IMI) (7) Simplicity (Simp) (8) Loose Coupling (LC) (9) Resource Discovery (RD) (10) Secured Communication (SC). The secured communication is a requirement that has always an inherently weak coverage, since at each level of the stack implementation or design solution, every approach has its own workarounds. In general, it is difficult to build HTTP client and server libraries because they are used in unexpected ways. Also the Apache *HTTP Components*, project claimed the excessive simplicity and flaws of most of the available HTTP client libraries, which causes both less code reuse and the proliferation of free and commercial independent HTTP clients. This leads the Internet protocols to additional complexity while covering any of the requirements.

The main gaps in the RESTful principles and Internet protocols, are all bridged by the specific IoT protocols implementations. However, a more efficient REST design, could further improve the performances. In particular, the RESTful architectural principles miss to explicitly address the reliability issue, the neighbor discovery and asynchronous interaction, in a resource-efficient way. Finally the resource discovery through the hypermedia representation, should be optimized, as explained in Sect. 3.3.5.

3.5 Scenario: CoAP and HTTP Heterogeneous Network

The typical IoT setting interconnects both sensor networks where either proprietary or CoRE protocols are implemented, with the Internet and its protocols. A central role in this setting, is played by proxies which, connected with proper backends,

enable transparent communication between ad-hoc, CoAP and HTTP networks. A network manager who wishes to develop such a proxy—although some open source implementations are already available[1]—should make the following considerations. Simply compressing HTTP would not be a solution because it would still retain the complexity and rely on the underlying TCP, in addition to the CPU cost required to perform the compression and the consequent gain of available bytes. CoAP supports a subset of the HTTP method, compliant to the RESTful principles. This is the key to enable the integration between resource constrained devices and the Web, as in the IoT. Then the proxying between CoAP and HTTP (as depicted in Fig. 3.17) is the most interesting one; although CoAP can be equally proxied to other protocols, e.g., XMPP, SIP. Only the Request/Response model of CoAP is mapped to HTTP, while the underlying messaging model must not be affected.

The communication between clients and servers is then intermediated by the proxy, which forwards requests and relay back responses; eventually performing caching, namespace or protocol translations. It can be classified either as a forward-proxy in case it performs requests on behalf of the client; as a reverse-proxy, in case it satisfies requests on behalf of one or more servers; or as a cross-proxy, in case it translates between different protocols. To avoid request loops, a proxy must be able to recognize all of its server names, aliases, local variations and IP addresses. This communication is based on messaging. CoAP allows addressing GET, POST, PUT and DELETE requests for resource constrained nodes. The GET method is both *safe* (i.e., the resource state on the server remains unchanged regardless of how many time the same URI is invoked) and *idempotent* (i.e., the resource state on the server remains unchanged regardless of how many times the same operation is repeated) and retrieves a representation for the information that currently corresponds to the resource identified by the request URI. The format of the representation is specified in the Content-Type header (otherwise it has to be derived from the specific application) and the preferred one can be set in the Accept request header. The POST method is neither safe nor idempotent and can result in either the creation, the update or the deletion of a resource. In case a resource is created, its URI should be specified in a sequence of one or more Location-Path or Location-Query options. The PUT method is not safe but it is idempotent. In fact, invoking the same URI more than once, would cause a not existing resource to be created, but an existing one to be only a modified version of the existing one. Finally, the DELETE method causes a resource to be deleted, if existing, this it is not safe, but it is idempotent. For seamless integration, reverse proxies can convert 6LoWPAN to IPv6 and CoAP/UDP to HTTP/TCP, so that sensor data can be accessed by using these omnipresent protocols. Also, Internet-based clients could directly use CoAP on top of UDP [29]. Despite the current gaps in REST, because of CoAP sticking to the RESTful principles, the cross-communication between heterogeneous networks is only a matter of proper proxy implementations.

[1] http://code.google.com/p/jcoap/

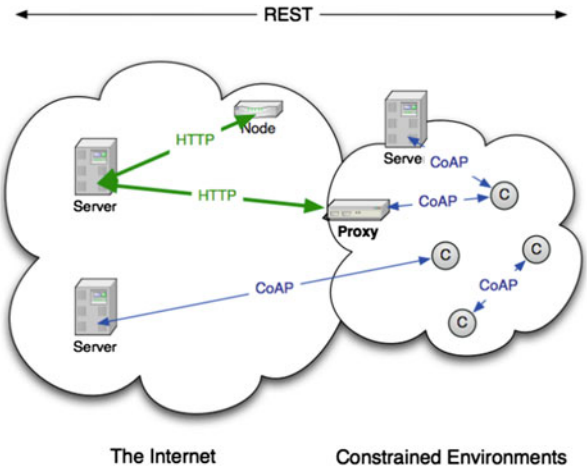


Fig. 3.17 The Embedded Web: architecture of a Cross-protocol proxying between HTTP and CoAP

3.6 Conclusion

We are currently witnessing a revolution in the interaction between humans and the real world. If we consider that we can only control what we can measure, then the growing amount of embedded devices by sending continuous streams of phenomenon (e.g., odor, light, noise, proximity) measurements are giving an augmented control to the people over the reality we live in. Combining these streams of data on the Internet of Things is however necessary, in order to grant a seamless data access and improve the data quality. Due to tight resource constraints on such devices, this too early claimed realization of the IoT vision, has not happened yet. However the research has moved further and, only a few months ago, an IETF WG finalized the 6LoWPAN, network layer protocol that unlocks IP-enabled devices, so that they can be directly connected to the Internet, as in the IoT vision. At the same time, ongoing IETF standardization efforts are directed towards an application protocol, CoAP, that is compliant with both resource constrained environments and RESTful principles' requirements. We painted a picture of the transformations that the Internet as-we-know-it is undergoing and the eventually necessary evolution of its underlying architectural design, REST. From a protocol stack perspective, we show which successful traditional Internet and IoT protocols' approaches can be leveraged, and which others should be addressed for their weaknesses. We focus on the successful features of the RESTful Web architecture on one side, and on its application in constrained environments on the other; discussing the main limitations. The current research direction seems the right one toward the enablement of a global nervous system of Internet-connected things, humans, animals and plants, which would increasingly resemble childhood fairy tales, where even a tree could speak[2].

[2] http://hello-tree.com/

Acknowledgements The authors would like to thank Zach Shelby, Prof. Dr.-Ing. Carsten Bormann, Prof. Dr. Elgar Fleisch and Lisa Dusseault for the inspirational content published online and for the permission to include their images. This work is funded by the Science Foundation Ireland (Grant No. SFI/08/CE/I1380 - Líon-2) and by the European Union (Grant ICT-258885 - SPITFIRE).

Chapter 4
Enabling Real-Time Resource Oriented Architectures with REST Observers

Vlad Stirbu and Timo Aaltonen

4.1 Introduction

The Web is build upon a simple interaction pattern. The web browser initiates a request to a resource and the web server responds with a representation that contains the current state of the respective resource. The web browser can check at a later time if the resource state has changed by making a new request and comparing the newly received representation against the previous one.

This basic model is inefficient and expensive for both browser and server, and valuable resources are wasted in the process. For example, the web server can be overloaded with requests to resources that have not changed while the users have to request new representations of resources of interest, and visually check in the browsers if there are changes. Additionally, because the refresh operation sends the entire representation of the resource, network bandwidth is used unnecessarily. Web caching and syndication related technologies mitigate some of these inefficiencies by reducing the frequency of requests, or the need to transfer representations. However, the interaction is still limited to a request-response pattern and the server has no means to initiate pushing changes to clients, when they occur.

In this chapter we present the REST observer, a framework for observation and notification that allows web clients to receive real-time events about state changes in resources of interest. First, we present the established mechanisms that improve interactiveness on the web. Second, we describe the REST observer pattern and the implementation in the Web context. Third, we analyze the qualities of the proposal in the context of the REST architectural style [81], and how such a system can be used in practice.

---

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

V. Stirbu (✉)
Mannikonkatu 4 A 3, Tampere 33820, Finland
e-mail: vlad.stirbu@ieee.org

T. Aaltonen
Tampere University of Technology, Korkeakoulunkatu 10, FI-33720, Tampere, Finland
e-mail: timo.aaltonen@tut.fi

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_4,

4.2 Motivation and Related Work

In this section we describe the main mechanisms that facilitate more efficient and interactive interactions between web clients and servers.

4.2.1 Web Syndication and Search Engines

Search engines were one of the first categories of user agents that consumed large amounts of information exposed on the web by automatically following the links embedded in web documents, the information being gathered to create indexes. The index is updated regularly by repeating the crawling process. To facilitate the process, web sites maintainers publish a Sitemap document [214] containing the list of resources that changed, when the change happen, how often it changes and how important document are in relation with other documents on the site. On another track, end users or web sites that aggregate content from third parties need more information when web documents change. They typically follow specific topic and are interested in a *summary* of the change that allows them to decide if the content should be retrieved. These updates are packaged as an RSS [201] or Atom [172] list, each entry being time-stamped, allowing clients to track back what happened on the site since the last visit. For large content publishers, polling their feeds is still expensive. To reduce the load, they publish feeds of feeds, like Simple Update Protocol (SUP) [40], which allows consumers to detect which feed is updated using a single operation. Although not proper push protocols, these mechanisms speed up considerably the process of propagating changes, while reducing the publishing costs.

4.2.2 Server-to-Server Change Propagation

PubSubHubbub (PuSH) [83] is a server-to-server protocol that notifies in close to real time when published content is updated. PuSH extends RSS and Atom protocols so that publishers inform consumers when the content changes eliminating the need for polling. The protocol involves the producer, the consumer and the hub, with an interaction that follows the publish/subscribe pattern. First, the consumer fetches the feed from the producer. The response contains a link to the hub that can deliver updates. Then, the consumer expresses interest in the respective feed to the hub. Whenever the content changes, the producer notifies the hub, which fetches the new content and makes it available to the consumer. Functional Observer REST (FOREST) [56] proposes a mechanism that allows web resources that depend on the state of other resources to be notified when the state change so that they can update their own state. The protocol that propagates the changes has an interaction pattern that is similar to PuSH.

4.2.3 Server-to-Client Change Propagation

The simplest method for a client to find out if a web resource has changed is to regularly request, or poll, the representations of the respective resource. The client checks the received representation against the previous known one and determines if there was a change. Long-polling is a more elaborate mechanism in which the server does not provide a representation until a change happens. Once the representation is received, the client repeats the procedure and waits for the next change. Alternatively, the server might choose not to close the connection and stream new data as it becomes available. These techniques are employed typically in web browsers through Asynchronous JavaScript and XML (Ajax) [158]. Alternatively, a server can push messages to a web browser using Server-Sent Events [110], which are typically streamed to the recipients. To receive these events, the client must have prior knowledge of the resource emitting the events, or becomes aware of the respective resource via a script.

WebSocket is technology that enables interactive communications between the browser and a web server. The clients can send messages to the server and receive responses without having to poll the server for a reply. Although, it simplifies the complexity around bi-directional web communication and connection management, it relies on an entirely new WebSocket protocol [75] that uses HTTP only during handshake.

4.2.4 Summary

We listed above the mainstream mechanisms that can be considered enablers for propagating changes in a resource' state. We can observe a group of mechanisms that are able to deliver the push-like notifications in real time, such as PuSH and FOREST, that require server functionality on the recipient end. On another hand, the Server-Send Event delivers the events to clients, but the expressiveness of the events it limited (e.g. they are text/event-stream messages), and the client has to have prior knowledge of the event emitting resource.

Our work is inspired by the functionality of the WATCH method proposed in ARRESTED [120], which extends the REST architectural style to support distributed and decentralized systems. However, unlike defining a new method, we wanted a push-like notification mechanism that can operate within the boundaries of the HTTP 1.1 [80] protocol.

4.3 The REST Observer Architecture Pattern

In this section we describe a pattern that enhances the REST architectural style [81] with event observation and notification capabilities. To realize these features we rely on the Observer pattern [90]. The proposed architecture is an instantiation of the internet-scale event observation and notification framework [200]. We are going

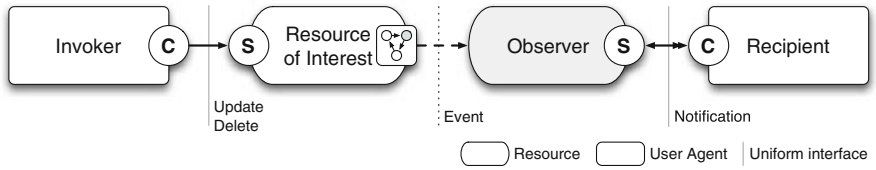


Fig. 4.1 The REST observer architectural model

through each of the models introduced by this design framework and describe how each concept is mapped to the REST observer architecture.

The Object Model The REST architectural style consists of clients and servers that interact using a uniform interface consisting of create, read, update, and delete (CRUD) operations. In this environment, an *invoking* user agent uses one of the operations provided by the uniform interface to acquire a representation corresponding to the current state of a resource that exposes the *object of interest*. For the remainder of the paper we will identify the object of interest as the *resource of interest*. Additional user agents interested in the state of the object of interest are the notification *recipients*. The architecture model of the REST observer is depicted in Fig. 4.1.

The Event Model The uniform interface allows the user agents to create a new resource, to read the current state of a resource, to update the resource determining a change in its state, or to delete the resource. Although all these operations trigger events, we are interested only in those events that occur due to changes in the representations of the resources of interest, or changes in relationships with other resources. Therefore, we consider *RESTful events* only those events that are triggered by a *successful* completion of an update or delete operation on the resource of interest.

The Naming Model The naming model used in our system is based on Uniform Resource Identifiers (URIs) [26]. Therefore, following the REST architectural style, the resources of interest are identified using unique URIs, while the user agents are identified by the identification of the user on which behalf the user acts, or anonymously, if the user agent is not identifiable.

The Observation Model The observation of events is realized using special resource type called *observer*. In the REST observer architecture we employ *synchronous* observation, event occurrences being explicitly communicated to the observers. The information policy mandates that event-specific information must contain enough data to allow the observer to recover the current state of the resource of interest. The minimal observer implementation does not enforce any observation policy, allowing an observer implementer to select the appropriate mechanism, for its environment, that achieves event observation.

The architecture does not specify a pattern abstraction policy or a filter policy, recipient user agents being notified about all individual events, as the observer does not filter any events and does not recognize any event patterns. The partitioning policy is considered an implementation detail of the resource of interest. Additionally, a minimal implementation of the observer is not required to maintain observations history.

The Time Model The REST observer architecture does not assume the existence of a global clock. Each event occurs in the context of a resource of interest and the local clock of that resource is used.

The Notification Model The communication between the recipient and the observer has two components. Initially, the recipient expresses interest in receiving events using the uniform interface of the observer. Then, the observer delivers notifications of every occurrence of an event to all recipients that have expressed the interest for as long as the recipients maintain the interest. The observer is the publisher and the recipient is the subscriber.

The Resource Model The resource of interest, the invoking user agent and the recipient user agent's lifetime does not depend on the occurrences of events or notification deliveries. In the resource model we are interested in the lifecycle of the observer. As the observer is a resource identifiable using a unique URI, we define that an observer is initiated with regard to a resource of interest if there is a relationship between the resource and the observer. Similarly, the observer is terminated if the relationship is ended. The existence of the relationship is conveyed in the representations provided by the resource of interest. The REST observer architecture does not define any management mechanism related to the lifecycle of the observer resource itself.

4.4 The Web-Based Realization of REST Observer

In this section we describe the implementation of the REST Observer pattern for the World Wide Web. We adapt the architecture pattern to the Web architecture [112], and provide examples of how to implement the mechanism using the HTTP protocol.

4.4.1 The Resource of Interest

A resource that has the ability to notify recipient user agents that its state or relationships with other resources have changed, can indicate such a capability by including the Link header [169] with a monitor [197] value in the response of the GET request:

```
#Request
GET {resource} HTTP/1.1

#Response
HTTP/1.1 200 OK
Link: {observerURI}; rel="monitor"

<!-- Resource representation -->
...
```

A user agent that understands the semantics of the Link header and the monitor relationship may decide to connect to the provided URI to receive notification related

to changes in resource' state. Some resources may include the relationship with the observer resource using representation specific mechanisms, such as the LINK header in the head section of HTML documents, or the link element in Atom documents[1]. The user agents that do not understand these mechanisms would ignore the resource relationship with the observer.

4.4.2 The Observer Resource

This resource provides the observer functionality. Because the state of the observer resource changes only over the event observation channel, user agents can only read the state of the resource.

4.4.2.1 Notification Delivery

The notifications are streamed by the observer resource, to each connected user agent, as entities within a single chunked encoded multipart/mixed message-body. Each notification must contain the HTTP protocol version, the status code and at least the Content-Type and Last-Modified headers.

```
# Request
GET {observerURI} HTTP/1.1

# Response
HTTP/1.1 200 OK
Content-Type: multipart/mixed;
              boundary="boundary-string"
Transfer-Encoding: "chunked"

--boundary-string
HTTP/1.1 {statusCode} {reasonPhrase}
Content-Type: {mimeType}
Last-Modified: {httpDate}
...

<!-- An optional message body -->
...
--boundary-string
HTTP/1.1 {statusCode} {reasonPhrase}
Content-Type: {mimeType}
Last-Modified: {httpDate}
...

<!-- Another optional message body -->
...
```

[1] Conveying the resource's relationship with the observer in the representation is the preferred method if the user agent is a browser that interacts with the observer using Ajax requests. In this environment, the browser engine implementation might not grant access to the HTTP headers included in the response, if the request was cross origin.

The value of the Last-Modified header conveys to the recipient user agent the time when the resource of interest changed state, while the Content-Type header describes if the notification is delivered inline or external.

4.4.2.2 Inline Delivery

An inline delivery notification contains the response to an HTTP request as if the notified user agent has made a GET request on the monitored resource. The inline notification must indicate the media type of the enclosed message body and the URI of the corresponding monitored resource, using the Content-Type and Content-Location headers:

```
...
--boundary-string
HTTP/1.1 200 OK
Content-Type: {mimeTypeOfTheMessageBody}
Content-Location: {monitoredResourceURI}
Last-Modified: Fri, 2 Nov 2012 22:16:08 GMT

<!-- Message body -->
...
--boundary-string
...
```

The notification is equivalent with the following HTTP request-response interaction:

```
# Request
GET {monitoredResourceURI} HTTP/1.1

# Response
HTTP/1.1 200 OK
Content-Type: {mimeType}
Last-Modified: Fri, 2 Nov 2012 22:16:08 GMT

<!-- Message body -->
...
```

4.4.2.3 External Delivery

An external delivery notification contains enough information to allow the recipient user agent to find out the latest state of a monitored resource using a subsequent GET request. The notification is represented as a message having the media type message/external-body [86]. The value of the encapsulated Content-ID header indicates the URI of the resource of interest being monitored. The notification itself does not include an actual body:

```
...
--boundary-string
HTTP/1.1 204 No Content
Content-Type: message/external-body;
              access-type=http

Content-ID: {monitoredResourceURI}
Last-Modified: Fri, 2 Nov 2012 22:55:08 GMT

--boundary-string
...
```

The recipient user agent of an external notification message may decide not to follow the URL indicated by the encapsulated Content-ID header, if its interest is limited only to knowledge that the resource has changed.

The server uses chunked transfer encoding to deliver the notifications to the clients. As the server can switch back and forth between inline and external deliveries without prior agreement, a client must understand both delivery mechanisms to work properly.

4.4.3 Notification Semantics

Each notification contains information that conveys to the recipient user agent the state of the monitored resource. The response status code and the message indicate the nature of the change, such as the resource of interest was updated or deleted. Optionally, a notification with inline delivery contains also the latest representation of a resource.

4.4.3.1 Updating or Editing

A resource is updated whenever a user agent successfully completed a PUT, a POST, or a PATCH [63] request on the monitored resource. The recipient user agent is not able to distinguish if the state change is a result of a partial or a complete update.

```
...
--boundary-string
HTTP/1.1 200 OK
Content-Type: {mimeType}
Content-Location: {updatedOrEditedResourceURI}

<!-- Message body -->
...

--boundary-string
...
```

4.4.3.2 Deletion

A resource is deleted whenever a user agent successfully completed a DELETE request on the monitored resource. The information is conveyed using the 410 response code and the status message Gone. The behavior is similar as if the recipient user agent would have made a GET request on the deleted resource of interest.

```
...
--boundary-string
HTTP/1.1 410 Gone
Content-Location: {deletedResourceURI}

--boundary-string
...
```

4.4.3.3 Creation

A user agent cannot subscribe to non-existing resources. To be able to monitor a resource, the user agent must find first the existence of the new resource. This can happen either by monitoring a collection to which the new resource is a member or by receiving a link to the resource in an update from an existing resource. We call this *indirect monitoring*.

4.4.4 Interaction

The typical interaction between the user agents and resources involved in the REST observer is initiated by the recipient user agent, see Fig. 4.2. The user agent makes a request on a specific resource and receives in the representation or as a header the URL where the resource can be monitored. Then, the user agents initiate the connection to the observer. To accommodate any possible changes in state in the resource of interest since the user agent received the response for the resource of interest since the connection with the observer was established, the observer sends a notification containing the last known state of the resource of interest. Later on, new notifications are delivered to the user agent as the observer becomes aware of changes in the resource' state.

4.5 Implementation Experience

To gain first hand experience with the REST observer pattern we created a prototype environment for a multiplayer turn-based game. The state of the game is maintained on a server while the players engage in the games using their mobile devices. As players complete their turns we want the other players to be notified about the game progress without having to poll the server.

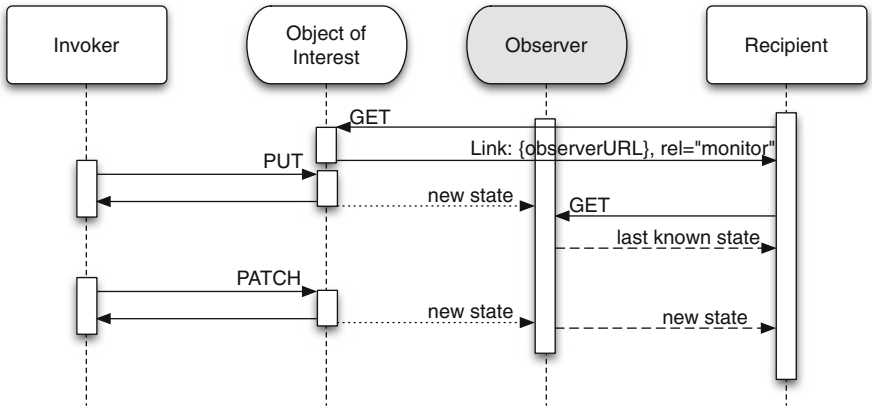


Fig. 4.2 Interaction example

Table 4.1 Model-resource mapping

Model	Resource
User	/user[a]
Game	/games/{gameId}
Turn	/turns/{turnId}
Observer	/observer[a]

[a] mapped based on successful authentication

4.5.1 The Prototype Environment

The server is implemented using express.js[2], a simple and flexible web framework running on node.js[3] that enables rapid creation of RESTful web applications. The client functionality is implemented as a single-page web application that is packaged as a native application using PhoneGap[4] framework. The application interacts with the server using Asynchronous JavaScript and XML (AJAX) requests.

4.5.2 Resources and Representations

The system is build around the following four resources: user, game, turn, and observer (see Table 4.1). The user resource exposes the games the user is involved in. The game resource is responsible for managing the life-cycle of games. For example, a user agent interacting with the resource can inspect that current state of a game, represented as an ordered list of turns, or can create new games. Whenever a new game is created the user resource of the players involved in the games is updated. The turn resource manages the game turn. A player can advance the game by filling in the turn requirements specific to the corresponding game type. Advancing to a new turn propagates to the associated game resource that updates its state, the time-stamp

[2] http://expressjs.com
[3] http://nodejs.org
[4] http://www.phonegap.com

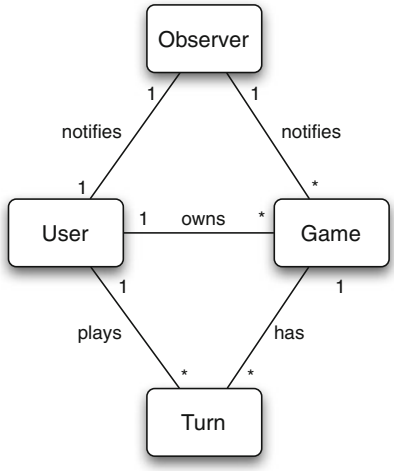


Fig. 4.3 Domain model used in prototype environment

being changed to the time the last turn was played. For brevity, the system maintains one observer resource for each user of the system. Therefore, a single observer resource will deliver notifications for all resources associated with the particular user. The domain model used in the prototype environment, including the resources, the relationships between the resources, and cardinalities of the relationships is depicted in Fig. 4.3.

The representation of the *user* resource is a Collection+JSON document [12]. This document conveys to the client the nature of the collection, using the profile relationship, and the where the resource can be monitored:

```
{
   "collection": {
      "version": "1.0",
      "href": "/user",
      "links": [
         {
            "rel": "monitor",
            "href": "/observer",
            "prompt": "Observer"
         },
         {
            "rel": "profile",
            "href": "/profiles/games",
            "prompt": "Profile"
         }
      ],
      "items": [
         {"href": "/games/{gameId}"},
         ...
      ]
   }
}
```

The client application becomes aware of the games played by the user it can retrieve the representations of individual games. Each *game* representation is a collection of turn items:

```
{
   "collection": {
      "version": "1.0",
      "href": "/games/{gameId}",
      "links": [
         {
            "rel": "monitor",
            "href": "/observer",
            "prompt": "Observer"
         },
         {
            "rel": "profile",
            "href": "/profiles/turns",
            "prompt": "Profile"
         }
      ],
      "items": [
         {"href": "/turns/{turnId}", data: [...]},
         ...
      ]
   }
}
```

4.5.3 Notifications Mechanics

When the application retrieves the representation of user and game resources it knows that it can monitor their state using the observer resource. Because the frequency of updates is relatively low, we decided to implement only the inline notifications.

When a new game is created, the user resource of all users invited to the game gets updated. As a result a notification is delivered through the observer to all connected user agents:

```
...
--boundary-string
HTTP/1.1 200 OK
Content-Type: application/vnd.Collection+JSON
Content-Location: /user

{
   "collection": {
      ...
      "items": [
         ...
         {"href": "/games/{newGameId}"},
      ],
      ...
   }
}
--boundary-string
...
```

Similarly, when a game turn is completed, a notification is delivered to all players involved in the game:

```
...
--boundary-string
HTTP/1.1 200 OK
Content-Type: application/vnd.Collection+JSON
Content-Location: /games/{updatedGameId}

{
   "collection": {
      ...
      "items": [
         {"href": "/turn/{firstGameTurnId}", data: [...]},
         ...
         {"href": "/turn/{completedGameTurnId}", data: [...]},
         ...
         {"href": "/turn/{lastGameTurnId}", data: [...]},
      ],
      ...
   }
}
--boundary-string
...
```

Because the application runs in mobile devices, we maintain the connection established with the observer only when the application runs in the foreground. When the application is switched to background the connection with the observer is closed. Whenever new updates are available the backend service pushes a notification to the mobile device using the operating system specific push mechanism, such as Apple Push Notification (APN) for iOS [17]. The payload size of the operating system specific notification is typically small: an APN notification cannot exceed 256 bytes, including the information on how the notification is presented to the user, therefore it can not carry the required payload. Whenever the device receives such a notification and the user decides not to ignore it, the application is moved in the foreground and re-establishes the connection to the observer resource.

4.6 Discussion

4.6.1 On the RESTfulness of the REST Observer Pattern

The REST architectural style consist of a set design guidelines that enable a network-based software system to have desirable properties such as "enhanced scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security and encapsulate legacy systems" [81].

By studying the REST observer pattern from this perspective, it is easy to notice that it follows the above design guidelines. The system is *client-server* with clearly separated responsibilities for the corresponding components. The interaction between the client and the server is *stateless*, as the recipient user agent interactions with the resource of interest and the observer happens in isolation. The data provided

in the notifications can be *cached* by the recipient user agent, each representation being valid until the observer notifies of a new change, or by the observer, which can deliver a change to multiple recipients. The components of the system interact using a *uniform interface*. The system is *layered*, as there might be intermediary observers that provide load balancing or shared-caching.

Additionally, the Resource Oriented Architecture (ROA) introduces a practical approach for describing the implementation of RESTful architectures, in the context of the World Wide Web, using four concepts (e.g. resources, URIs, representations, and links between the resources), and four properties (e.g. addressability, statelessness, connectedness and uniform interface). The REST observer introduces a new resource, the observer, that can be identified using a URI, delivering to the client well known representations for inline and external notifications. The observer resource is connected with the resource of interest that is monitored using web linking. The observer pattern integrates easily into a ROA system, as it provides also addressability of the observer resource, the interaction between recipient user agents and observer is stateless, the observer is connected with the resource of interest, and the interaction is done using the GET verb of the HTTP protocol.

Further, we can see that the REST observer obeys the *hypermedia as the engine of application state (HATEOAS)* principle, as a recipient user agent does not have to have prior knowledge of how to interact with the observer. It simply discovers if a particular resource of interest is able to deliver notifications, on its state change via an observer, by looking if the response headers contain web links or the representation has embedded links to the respective observer resource.

4.6.2 On the Implementation of the Uniform Interface

The REST observer pattern allows a user agent to be notified when a resource of interest state has changed. This goal is achieved using the observer resource and augmenting the GET interface of the resource of interest to include a link to the observer resource. The arrangement isolates the implementation of the resource of interest from the recipient user agent. Using only the GET method, the recipient user agent can observe the changes in state of the respective resource, while also discovering the observer resource that delivers the change notifications. The resource of interest implementer may decide to allow other methods, besides the ones defined by HTTP 1.1, to change the state of the resource. For example, we used PATCH [63] in our prototype environment to update the state of the game. However, regardless of how the resource state has changed, the recipient user agent receives the new state as if it would have made a GET request. Therefore, the implementation details of the resource of interest interface are not exposed to the recipient user agent over the observer resource interface.

The overall perceived behavior of the REST observer pattern is equivalent in functionality with the WATCH method proposed by ARRESTED. However, obtaining

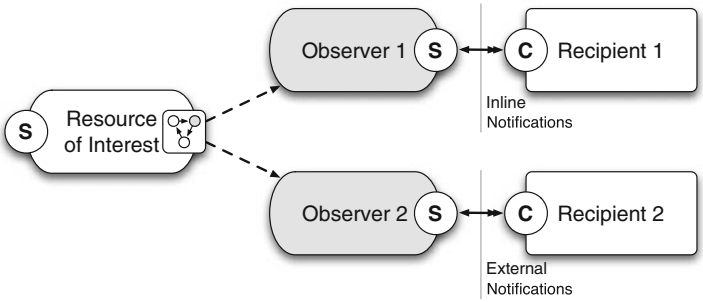


Fig. 4.4 Scenario with one resource having two observers

the same results by with the GET method has immediate deployment and operational advantages. For example, existing software tools can be leveraged to develop the REST observer and to augment already implemented ROA services. Additionally, HTTP intermediaries do not have adverse side effects as they are expected to understand and handle correctly GET and POST methods [1].

4.6.3 Observation Granularity and Filtering

The REST observer pattern enables two levels of descriptiveness for the delivered notifications. The inline notification delivers the representation that corresponds to the new state of the resource of interest, allowing a user agent to have an up to date state of the resource without further actions. On the other hand, the external notification informs the user agent that the resource has changed and when, but the user agent has to make a subsequent request to become aware of the state of the resource. This level of control enables a user agent interested only when the resource state changed to stay informed about this situation. Further, there are situations when a user agent is not interested in the full state of the resource but only in partial states. A resource that is able to distinguish the partial states can notify interested user agents using the URI query parameters or fragments.

4.6.4 Example Deployments

The relationships between the resources of interest and observers are an implementation specific. However, there are a few deployment scenarios that are worth discussing to emphasize how the observer pattern can be used in practice.

In the scenario presented in Fig. 4.4, there are a number of user agents that are interested to know only if the resource of interest has changed but not what is the latest state. To accommodate this situation, the resource of interest implementer decides to provide two observer resources: one delivers inline notifications, and the

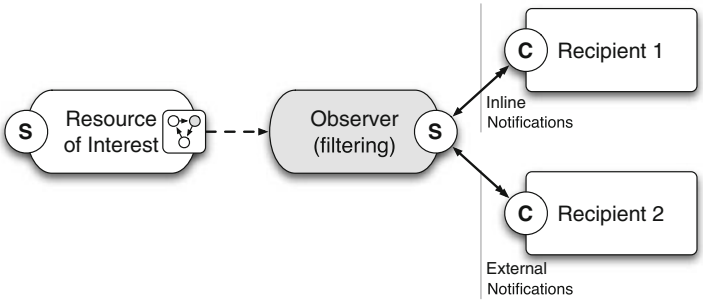


Fig. 4.5 Scenario with one resource having one observer

other delivers external notifications. The relationship is conveyed to the user agents, using web linking:

```
#Request
GET /resource HTTP/1.1

#Response
HTTP/1.1 200 OK
Link: </observer1>; rel="monitor"; type="application/json",
      </observer2>; rel="monitor"; type="message/external-body"
<!-- Resource representation -->
...
```

A similar result as above can be obtained using a single observer that has the ability to filter itself the inline and external notifications based on the preferences expressed by the recipient user agents. This environment is depicted in Fig. 4.5.

The resource of interest would inform the user agents on how to interact with the observer through web linking:

```
#Request
GET /resource HTTP/1.1

#Response
HTTP/1.1 200 OK
Link: </observer?notification=inline>; rel="monitor";
      type="application/json",
      </observer?notification=external>; rel="monitor";
      type="message/external-body"
<!-- Resource representation -->
...
```

A more elaborate scenario involves chained observers, in which downstream observers provide features that are not available upstream, is presented in Fig. 4.6. The resource of interest, aware of the observer chains, would inform the user agents on how to interact with the observer using web linking:

```
#Request
GET /resource HTTP/1.1

#Response
HTTP/1.1 200 OK
Link: </observer1>; rel="monitor",
      </observer2?notification=inline>; rel="monitor";
      type="application/json",
      </observer2?notification=external>; rel="monitor";
      type="message/external-body"
<!-- Resource representation -->
...
```

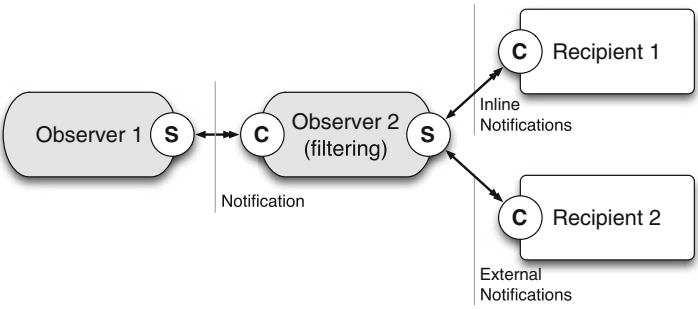


Fig. 4.6 Scenario with chained observers

4.7 Conclusions and Future Work

The REST observer pattern delivers push-like functionality that enables users agents interested in the state of a resource to stay updated when the state of the resource of interest has changed. However, instead of using entirely new protocols, like WebSockets, or define new methods as proposed in ARRESTED, we achieve equivalent results within the boundaries on the HTTP 1.1 protocol.

We plan to improve the REST observer by developing an observer life-cycle mechanism that allows observer resources to be created on demand, and an event filtering mechanisms that allow user agents to specify which events are of interest, so that observer delivers only those events. We also intend to investigate more complex deployment scenarios in which chained observers provide sophisticated event filtering. Additionally, the responsiveness of the system can be improved by delivering partial representations of new states, or event updates on partial states. For example, instead of delivering full-size JSON implementations, the server could deliver JSON Patch [39] documents that contain only the difference form the previous delivered state.

Chapter 5
Survey of Semantic Description of REST APIs

Ruben Verborgh, Andreas Harth, Maria Maleshkova, Steffen Stadtmüller, Thomas Steiner, Mohsen Taheriyan and Rik Van de Walle

5.1 Introduction

The REST architectural style assumes that client and server form a contract with content negotiation, not only on the data format but implicitly also on the semantics of the communicated data, *i.e.*, an agreement on how the data have to be interpreted [247]. In different application scenarios such an agreement requires vendor-specific content types for the individual services to convey the meaning of the communicated data. The idea behind vendor-specific content types is that service providers can

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

R. Verborgh (✉)
Multimedia Lab – Ghent University – iMinds, Gaston Crommenlaan 8 bus 201,
Ledeberg-Ghent 9050, Belgium
e-mail: ruben.verborgh@ugent.be

A. Harth · M. Maleshkova · S. Stadtmüller
Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe 76128, Germany
e-mail: harth@kit.edu

M. Maleshkova
e-mail: maria.maleshkova@kit.edu

S. Stadtmüller
e-mail: steffen.stadtmueller@kit.edu

T. Steiner
Departament de Llenguatges i Sistemes Informatics, Universitat Politècnica
de Catalunya, Jordi Girona, 29, Barcelona 08034, Spain
e-mail: tsteiner@lsi.upc.edu

M. Taheriyan
Information Science Institute, University of Southern California, Admiralty Way,
Suite 1001, Marina del Rey, CA 4676, USA
e-mail: mohsen@isi.edu

R. Van de Walle
Multimedia Lab – Ghent University – iMinds, Gaston Crommenlaan 8 bus 201,
Ledeberg- Ghent 9050, Belgium
e-mail: rik.vandewalle@ugent.be

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_5,

reuse content types and service consumers can make use of specific processors for the individual content types. In practice however, we see that many RESTful APIs on the Web simply make use of standard non-specific content types, *e.g.*, *text/xml* or *application/json* [150]. Since the agreement on the semantics is only implicit, programmers developing client applications have to manually gain a deep understanding of several APIs from multiple providers.

Common Web APIs are typically either exclusively described textually[1], or far less frequently—and usually based on third-party contributions—a machine-readable WADL [104] API description exists[2]. However, neither human-focused textual, nor machine-focused WADL API descriptions carry any *machine-processable* semantics, *i.e.*, do not describe *what* a certain API does. Instead, they limit themselves to a description of machine-readable in- and output parameters in the case of WADL, or a non-machine-readable prose- and/or example-driven description of the API in the case of textual descriptions. While this may suffice the requirements of developers in practice, the lack of semantic descriptions hinders many more advanced use cases such as API discovery or API composition.

Machine-interpretable descriptions can serve several purposes when developing client applications:

- One can generate textual documentation from the standardised machine-interpretable descriptions, which leads to a more coherent presentation of the APIs, similar to what JavaDoc has achieved in the Java world (see also Knuth's idea of literal programming [125]).
- A standardised way to access REST APIs introduce a higher degree of automation for high level tasks such as composition.
- Machine-interpretable descriptions facilitate a more structured approach to developing APIs, which means that automated tools can check coherence and the RESTful-ness of an API (e.g., according to the Richardson Maturity Model[3]).

With hypermedia being an essential constraint of the REST architectural style [78], the transitions between different application states of a REST API have to be hypermedia-driven. In that sense, semantic API descriptions complement the hypermedia-driven interactions by providing additional clues for machine clients that help them make informed decisions.

In a RESTful interaction with Web resources, only the constraint set of HTTP methods can be applied to the resources. The semantics of the HTTP methods itself is defined by the IETF [80] and do not need to be explicitly described for individual resources. We can distinguish between safe and non-safe methods, where safe methods guarantee not to affect the current state of resources. Additionally, some of the methods require additional input data to be provided for their invocation. The communicated input data can be subject to requirements that need to be described to allow an automated interaction. Furthermore, the effect on the resources state of an

[1] For example, the Twitter REST API: https://dev.twitter.com/docs/api

[2] A large archive of WADL descriptions is available on GitHub: https://github.com/apigee/wadl-library

[3] http://martinfowler.com/articles/richardsonMaturityModel.html

application of a non-safe method has needs to be assessable before the actual invocation to allow clients to decide how to interact with the resources. The effected change of resources after applying an HTTP method can also depend on the communicated input data. This dependency between communicated input and the resulting state of resources has also to be subject of a description.

In the chapter we classify the various approaches for providing machine-interpretable descriptions of Web APIs. Section 5.2 surveys lightweight semantic descriptions. Section 5.3 introduces descriptions based on graph patterns (a subset of the SPARQL query language for RDF, a graph-structured data format). Section 5.4 covers logic-based descriptions. Section 5.5 covers JSON-based descriptions. Section 5.6 contains a description of two tools for annotating existing APIs, and Sect. 5.8 concludes.

5.2 Lightweight Semantic Descriptions

5.2.1 Syntactic REST API Descriptions

Web services enable the publishing and consuming of functionalities of existing applications, facilitating the development of systems based on decoupled and distributed components. **WSDL** [49, 254] is an XML-based language for describing the interface of a Web service. A WSDL service description specifies: (1) the supported operations for consuming the Web service; (2) its transport protocol bindings; (3) the message exchange format; and (4) its physical location. In this way, the WSDL description contains all information necessary for invoking a service and since it is XML-based, conforming to the WSDL xml schema, it is also machine-processable.

WSDL Similarly, to Web services, which provide access to the functionality of existing components, Web APIs and Web applications conforming to the REST paradigm, provide access to resources by using the WWW as an infrastructure platform. Based on this parallel between the two types of services, WSDL was extended to Version 2.0 [254], which can also be used for formally describing RESTful services. As a result WSDL is a machine-processable, platform- and language-independent form of describing Web services and RESTful services alike.

The difficulty of using WSDL for RESTful services, is that it was not especially designed for resource-oriented services and as a result, everything has to be described in an operation-based manner. In addition, WSDL introduces some difficulties with specifying the message exchange format and limits HTTP authentication methods. Moreover, the most important drawback is that it lacks support for simple links. There is no mechanism in WSDL 2.0 to describe new resources that are identified by links in other documents. However, one of the most important characteristics of RESTful services is that they consist of collections of interlinked resources. Finally, the adoption of WSDL as means for describing Web APIs would requite that all providers update or completely change their websites with documentation, moving

away for using only text in HTML form. Similarly developers would need to learn to deal with WSDL instead of simply reading a natural language description.

WADL In contrast to WSDL, the Web Application Description Language (WADL, [104]) was especially designed for describing RESTful services in a machine-processable way. It is also XML-based and is platform and language independent. As opposed to WSDL, WADL models the resources provided by a service, and the relationships between them in the form of links. A service is described using a set of resource elements. Each resource contains descriptions of the inputs and method elements, including the request and response for the resource. In particular, the request element specifies how to represent the input, what types are required and any specific HTTP headers. The response describes the representation of the service's response, as well as any fault information, to deal with errors.

Currently, neither WADL nor WSDL are widely accepted and used for Web APIs and RESTful services. A Google search for WADL files returns only 49 unique results[4], while from the popular Web 2.0 applications only delicious[5] and YahooSearch[6] have WADL descriptions. The main difficulty of using WADL descriptions is that they are complex, in comparison to text-based documentation, and require that developers have a certain level of training and tool support that enables the application development on top of WADL. This complexity contradicts with the current proliferation of Web APIs, which can be greatly attributed to simplicity and direct used of the infrastructure provided by the Web, which enable the easy retrieving of resources only thought an HTTP request, directly in the Web browser.

Web APIs evolve rather autonomously without conforming to a shared set of guidelines or standards, especially evident by the fact the documentation is usually given in natural language as part of a webpage. The developer has to decide what structure to use and what information to provide. As a result, everyone who is able to create a Web page is also able to create a Web API description.

However, plain text/HTML descriptions, in contrast to WSDL and WADL descriptions, are not meant for automated machine-processing, which means that if developers want to use a particular service, they have to go to an existing description Web page, study it and write the application manually. Therefore, current research proposes the creation of machine-interpretable descriptions on top of existing HTML descriptions by using **microformats** [119]. Microformats offer means for annotating human-oriented Web pages in order to make key information automatically recognisable and processable, without modifying the visualization or the content.

hREST One particular approach for creating machine-processable descriptions for RESTful services by using microformats is hRESTS (HTML for RESTful Services) [129]. hRESTS enables the marking of service properties including operations, inputs and outputs, HTTP methods and labels, by inserting HTML tags within the HTML. In this way, the user does not see any changes in the Web page, however,

[4] Search done on October 5th, 2009
[5] http://delicious.com/
[6] http://search.yahoo.com/

based on the tags, the service can be automatically recognized by crawlers and the service properties can directly be extracted by applying a simple XSL transformation. The result is an HTML page that also contains the syntactical information of the described Web API and therefore, no longer relies solely on human interpretation. Versioning in hRESTS is dealt with the same way as in microformats through backwards-compatible additions of class names[7], potentially requiring wrapping elements.

RDFa An alternative to using hRESTS is offered by RDFa [2] that enables the embedding of RDF data in HTML. RDFa is similar to using microformats, but is somewhat more complex and offers more HTML markup options, as opposed to hRESTS. Approaches, based on making existing RESTful service descriptions machine-processable by using HTML tags are simpler and more lightweight as opposed to WSDL and WADL. In addition, as already mentioned, they can be applied directly on already available descriptions, rather then creating new service descriptions from scratch. The adoption by developers is also easier, since the creation of a machine-processable RESTful service description is equivalent to Web content creation or modification.

5.2.2 MicroWSMO/SA-REST

In contrast to research in the area of Semantic Web Services, which has been quite prolific, the number of semantic frameworks targeted at capturing Web API characteristics is relatively limited. Web APIs have only recently achieved greater popularity and wider use, thus raising the interest of the research community. In this section we discuss the two main approaches—MicroWSMO and SA-REST, aiming to support a greater level of automation of common service tasks through employing semantics. We also consider further description languages and ontologies, including ReLL and ROSM.

MicroWSMO MicroWSMO [128] is a formalism for the semantic description of Web APIs, which is based on adapting the SA-WSDL [73] approach. MicroWSMO uses microformats for adding semantic information on top of HTML service documentation, by relying on hRESTS for marking service properties and making the descriptions machine-processable. It uses three main types of link relations: (1) **modelReference**, which can be used on any service property to point to appropriate semantic concepts identified by URIs; (2) **liftingSchemaMapping** and (3) **loweringSchemaMapping**, which associate messages with appropriate transformations (also identified by URIs) between the serialization format such as XML and a semantic knowledge representation format such as RDF. Therefore, MicroWSMO, based on hRESTS, enables the semantic annotation of Web APIs in the same way in which SA-WSDL, based on WSDL, supports the annotation of Web services.

[7] http://microformats.org/wiki/microformats-2#backward_compatible

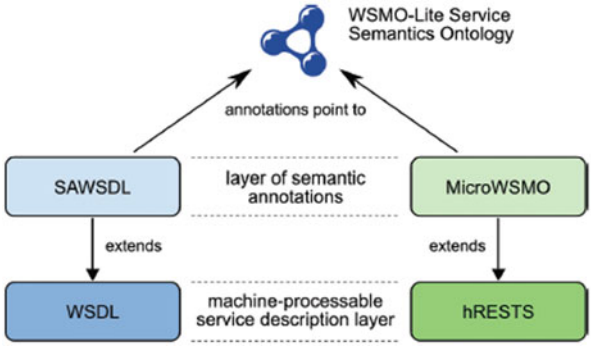


Fig. 5.1 Unifying SA-WSDL and MicroWSMO through WSMO-Lite

In addition, MicroWSMO can be complemented by the WSMO-Lite service ontology specifying the content of the semantic annotations (see Fig. 5.1[8]). Since both Web APIs and WSDL-based services can have WSMO-Lite annotations, this provides a basis for integrating the two types of services. Therefore, WSMO-Lite enables unified search over both Web APIs and WSDL-based services, and tasks such as discovery, composition and mediation can be performed based on WSMO-Lite, completely independently from the underlying Web service technology (WSDL/SOAP or REST/HTTP).

SA-REST Another formalism for the semantic description of RESTful services is SA-REST [212], which also applies the grounding principles of SA-WSDL but instead of using hRESTS relies on RDFa [2] for marking service properties. Similarly to MicroWSMO, SA-REST enables the annotation of existing HTML service descriptions by defining the following service elements: input, output, operation, lifting, lowering, or fault and linking these to semantic entities. The main differences between the two approaches are not the underlying principles but rather the implementation techniques. In addition, MicroWSMO aims to add semantics as means for Web service automation, while SA-REST is more oriented towards enabling tool support in the context of service discovery and composition by mashup or smashup [211].

5.2.3 Minimal Service Model

The Minimal Service Model (MSM) represents an operation-based approach towards describing Web APIs. It is a simple RDF(S) ontology that supports the annotation of common Web API descriptions. It also aims to enable the reusability of existing Semantic Web service approaches by capturing the maximum common denominator between existing conceptual models for services. Additionally, as opposed to most semantic Web services research up to date, MSM targets to support both traditional Web services, as well as Web APIs with procedural view on resources, so that they can be handled in a unified way.

[8] http://www.wst.univie.ac.at/workgroups/sem-nessi/index.php?t=semanticweb

Originally the MSM was introduced together with hRESTS [129], aiming to cover for the fact that Web API descriptions do not typically have any structure in terms of the resources handled or the operations exposed. Although its initial purpose was to provide structure to hRESTS, it has been subsequently adjusted and updated to its current form, in order to provide means for integrating heterogeneous services (*i.e.*, WSDLs and Web APIS), and together with WSMO-Lite it has been used as a means to facilitate a common framework covering the largest common denominator of the most used Semantic Web service formalisms on the Web. On the basis of MSM and WSMO-Lite, generic publication and discovery machinery has been developed that supports SA-WSDL, WSMO-Lite, hREST/MicroWSMO, and OWL-S services [186]. The Semantic Web Services exposed by this infrastructure [185] put the emphasis on reducing the complexity of conceptual models and integrating services with existing Linked Data [33]. This integration serves both as a means to simplify the creation and management of Semantic Web Services through reuse, as well as it provides a new view over semantic Web services understood as a means to support the generating and processing Linked Data on the Web. Subsequent work around the Minimal Service Model is focused on supporting the invocation and authentication of Web APIs [143], [149].

The original MSM [129] defined a Web API in terms of *Services* that has a number of *Operations*, which have an *Input*, an *Output*, and *Faults*. However, the original MSM, which was used as a basis for SA-REST [212] and MicroWSMO [128], fails to capture some significant parts of the descriptions, such as optional, default and mandatory parameters, which can have a crucial effect on discovery and invocation. In addition, it does not enable the description of the inputs, parts of the input, and parts of the parts of the input. As a result, the MSM has been extended [186] to support further service characteristics.

The MSM, in its current version [186], is visualized in Fig. 5.2. The MSM, denoted by the *msm* namespace, defines *Services* with a number of *Operations*. Operations in turn have input, output and fault *MessageContent* descriptions. A *MessageContent* may be composed of *MessageParts*, which can be mandatory or optional. The intent of the message part mechanism is to support finer-grain discovery based on message parts and allowing distinguishing between mandatory and optional parts.

The MSM is used as the core model for describing Web APIs, while supplementary models are defined for capturing details that are of particular relevance for the separate service tasks. As part of identifying extensions to the MSM, work on supporting the automation of authentication [149] was conducted.

5.2.4 Further Semantic Approaches

ROSM The Resource-Oriented Service Model (ROSM [82]) ontology is a lightweight approach to the structural description of resource-oriented RESTful services, compatible with WSMO-Lite annotations. It enables the annotation of resources belonging to a service. In turn the resources can be described as being

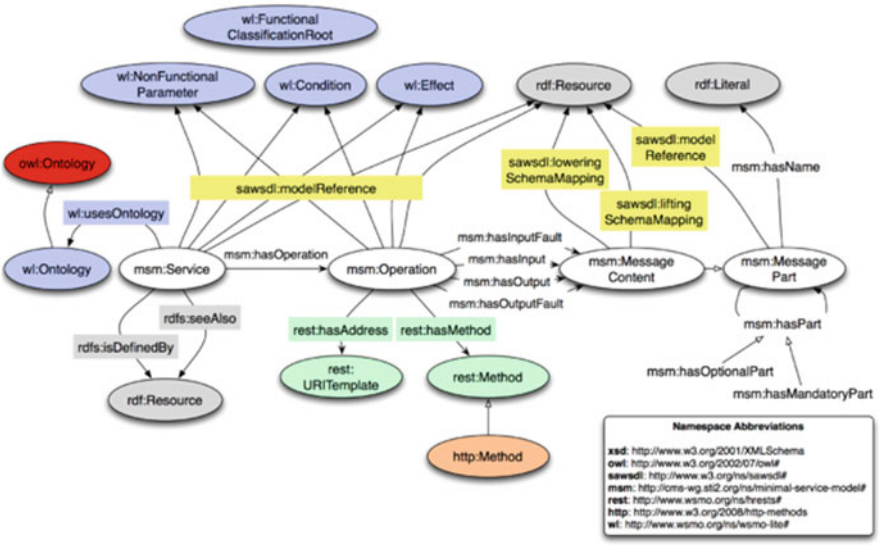


Fig. 5.2 Minimal service model

part of collections and having addresses (URIs) serving as endpoints for access and manipulation. The organization of resource in collections, which again belongs to a service, allows capturing an arbitrary number of resources and attaching service semantics to them following the SA-WSDL approach.

Furthermore, resources can have certain HTTP methods associated with them, which define how it is possible to interact with a resource, which are connected through an operation. These operations are modeled in a much-more fine-grained way, since they basically only have to support the uniform interface of HTTP. In addition, ROSM[9] enables the explicit modeling of requests and responses with their associated aspects (*e.g.*, parameters or response codes). In summary, ROSM represents a simple ontology for describing resource-centered services, in terms of resources, collection of resources, addresses and HTTP methods. The specific semantics of the ROSM annotations can be described by linking to the WSMO-Lite ontology.

SERVONT is an ontology-based hybrid approach where different kinds of matchmaking strategies are combined together to provide an adaptive, flexible and efficient service discovery environment [62]. Semantic-enriched frameworks are considered a key issue to enforce timely discovery and dynamic composition of services. In contrast, the ServFace project aims at creating a model-driven service engineering methodology for an integrated development process for service-based applications[10]. Finally, the Simple Semantic Web Architecture and Protocol (SSWAP) is the driving technology for the iPlant Semantic Web Program[11]. It combines Web service

[9] http://www.wsmo.org/ns/rosm/0.1/
[10] http://www.servface.eu/
[11] http://sswap.info/

functionality with an extensible semantic framework to satisfy the conditions for high throughput integration [91]. SSWAP originates from the Semantic MOBY project, which is a branch of BIOMOBY project [251]. Using SSWAP, users can create scientific workflows based on the discovery and execution of Web services and RESTful services.

5.3 SPARQL-Based Descriptions

Following the motivation to look beyond the exposure of fixed datasets, an extension of Linked Data with REST technologies has been proposed and explored for some time [24], [134], [168], [249], [217], [240]. These approaches extend the traditional use of HTTP in Linked Data, consistent with REST, by allowing all HTTP operations to be applied to Linked Data resources. REST services in this context are often perceived RDF prosumers, *i.e.*, the state of resources is made available encoded in RDF, at least as an alternative via Content Negotiation. Clients can interact with Linked Data resources by submitting RDF input data with HTTP methods (*e.g.*, POST, PUT) resulting in the manipulation or creation of resources. The output data a client receives after the successful submission of data describes the effected state change of resources (*i.e.*, their new content after creation or manipulation) and is serialized in RDF as well.

Such REST services contribute to the Web of Data by interlinking output data with existing Linked Data sets.

RESTful Linked Data resources consider how the data that results from computation over input can carry explicit semantics and base their service descriptions on the notion that Linked Data provides a description for resources' input and output requirements: the graph patterns provided by the SPARQL query language or the N3 notation. Graph pattern provide the advantage of a more thorough description of what should be communicated, familiarity to Linked Data producers and consumers, and the possibility for increased tool support.

The rational behind the descriptions for RESTful Linked Data resources is that the current state of a resource can be retrieved with an HTTP GET, while the data exchange that constitutes a manipulating interaction with a resource is described with two graph patterns:

- A graph pattern that describes the RDF input data that is submitted to a resource (*e.g.*, with HTTP POST), which is necessary to invoke the manipulation.
- A graph pattern that describes the RDF output data the client receives after a successful call.

The description implies that the input pattern has to match the input data to invoke the service and the output pattern will match the output data returned by the service.

To illustrate the use of graph pattern-based IO descriptions we introduce an example based on an API of a social network platform. A common feature of social networks is to post a message to the timeline of a user. The approach in the context

Table 5.1 Example description for a timeline resource in a social network

Input:	{ ?post a sioc:Post. ?post sioc:content ?content. ?post sioc:has-creator ?user. }
Output:	{ ?post sioc:content ?content. ?post dcterms:created ?date. }

of Linked Data would be to wrap this call to provide the information in RDF, reusing existing vocabularies. Here the natural choice is the SIOC vocabulary[12].

The description of a resource representing the timeline of a user, would make explicit as required input for an HTTP POST call the content of the message and its creator identified in that social network. Furthermore the output would be described as the created message with enriched with additional information (*e.g.*, the creation date). A client would receive data matching this pattern together with the HTTP status code indicating the success of the call. The resulting input and output descriptions for the Linked API are represented in Table 5.1.

Note that the reuse of the variables ?post and ?content across input and output implies that this variable will have the same binding in the output as provided in the input. In SPARQL terms these would be 'safe variables' if we considered the service to be a SPARQL CONSTRUCT query from the input to the output patterns. Variables in the output pattern that do not appear in the input pattern are bound by the functionality of the resource, *i.e.*, the binding is a result of the manipulation or creation of a resource.

Graph pattern based descriptions allow for a thorough descriptions of what a client has to communicate to successfully interact with a RESTful Linked Data resource. The output descriptions that share variables with the input descriptions allow clients to anticipate the result of their interaction with respect to the input they intended to provide. Such an anticipation is due to the circumstance that the actual output message of an interaction is intended to convey the effected state change of an interaction. The predictability of effects of manipulating actions is essential to enable (semi-)automated clients that use Linked Data REST services.

5.4 Logic-Based Descriptions

Since its inception, the Semantic Web has always had a strong link with logic [27]. The impact of logic is visible in essential building stones such as RDF [123], whose *open world* assumption prohibits conclusions based on the absence of certain triples. Two logic families are predominant on the Semantic Web: *description logic*, the underlying model of OWL [156], is the most widespread, followed by *first-order logic*, which is typically expressed in extensions of RDF with rules. Description logic is in essence a decidable fragment of first-order logic, at the cost of a loss in expressivity.

[12] http://sioc-project.org/ontology/

This reduced expressivity is a motivator for LOS and LIDS to adopt SPARQL and the reason for a few other methods use rule-based expression languages. For example, in OWL-S [153], one of the early semantic description formalisms for traditional Web services, rule languages such as KIF or SWIRL have to be used to express various aspects that extend beyond the expressivity of RDF. These languages capture expressions for pre- and postconditions, results, and effects of a Web service invocation. In general, rules are a straightforward mechanism to express dynamic relationships, such as those that occur with Web APIs.

5.4.1 RESTdesc

RESTdesc [240], [241] is a logic-based description method that focuses on exposing the functional aspect of Web APIs in machine-processable form. Concretely, RESTdesc descriptions are rules expressed in the Notation3 language (N3 [25]), which adds support for variables and quantification to RDF. The latter is a prerequisite to natively describe statements such as "all requests to y result in x", which is not directly supported in RDF (*i.e.*, only through the use of modeling vocabularies). In fact, RESTdesc uses quantification to express functionality as follows:

$$\forall x \quad preconditions(x) \Rightarrow \exists r(request(r, x) \wedge postconditions(x))$$

Herein, the predicates are expressed as RDF graphs, called *formulae* in N3. Reasoners interpret the above as "for every situation x where certain preconditions are met, there exists a specified request r that makes certain postconditions true." Moreover, for specific situations x_n, reasoners can instantiate the above rule, thereby creating an RDF representation of a concrete HTTP request r_n, which can be executed by any RDF-compatible HTTP client.

Descriptions can be used to express the effects that occur as a result of a POST request, including the description of that request itself. Representations are *not* described, as RESTdesc aims to be representation-independent, but it can describe properties of these representations. This is not unlike the graph patterns used in SPARQL-based methods (see Sect. 5.3), but represented in a syntax that integrates the request description. RESTdesc descriptions can also be useful to describe requests with safe methods such as GET, in order to explain what properties the response will satisfy, again without constraining the representation. This can be useful in two cases: (*a*) if the client does not want to retrieve the resource (for instance, if there are too many) or (*b*) if the resource does not exist yet, but can be created as the result of another action. In both cases, clients can then predict certain properties of the resource, without actually accessing it.

Since RESTdesc descriptions are expressed as rules, they inherit the logical functionality, in particular the *chaining* property:

$$P \Rightarrow Q,\ Q \Rightarrow R \quad \vdash \quad P \Rightarrow R$$

Therefore, existing Semantic Web reasoners with N3 support can create compositions of RESTdesc-described Web APIs by chaining RESTdesc descriptions [238], either

in a forward or backward (goal-driven) manner. This enables agents to respectively discover possible actions and to find a sequence of actions to satisfy a predetermined goal.

RESTdesc descriptions have been designed [240] to support hypermedia-driven applications [78], as they essentially enable machines to anticipate on possible controls a resource might have. Client still need to operate as hypermedia consumers by following links, but the descriptions allow them to predict beforehand what resources can be of interest. For example, understanding that a POST request on a certain resource will lead to the creation of a subresource with a certain affordance, allows to client to reason about whether this action is desired.

Further work on RESTdesc includes the incorporation of quality parameters, possibly subjective, in Web API descriptions [239].

5.5 JSON-Based Descriptions

So far we have explored solutions that are directly based on Semantic Web technologies. However, many Web developers are reluctant to integrate RDF, SPARQL or N3 in their applications. Lanthaler and Gütl provide three reasons for what they call *semaphobia* [137]. Firstly, they observe that the perceived complexity of the Semantic Web, in combination with its background in artificial intelligence, makes developers assume integration will be difficult. Furthermore, some of them are unsure whether the integration cost will be worthwhile. After all, the Semantic Web is sometimes regarded as a solution in search for a problem, suffering from the chicken-and-egg syndrome and thus still waiting for a *killer application* (although the W3C maintains a list of case studies and use cases [20]). Finally, the Semantic Web is often incorrectly considered a disruptive technology that is hard to implement in an evolving ecosystem [215].

Whatever the reasons for semaphobia, adoption is often a decisive factor for technologies. Therefore, description formats anchored on a technology Web developers are already familiar with have a head start. As of 2012, more than 44 % of all APIs on ProgrammableWeb, the largest API index [64], communicate in JSON. JSON is the JavaScript Object Notation language, which quickly became popular on the Web as it was natively parsed by JavaScript, the only scripting language supported by the majority of browsers. Its simplicity and extensibility make it also an interesting target for many other environments [207].

5.5.1 SEREDASj

As the letter *j* in its name suggests, SEREDASj (*s*emantic *re*stful DAta services [137]) is a semantic description language format expressed in JSON. The motivation behind

SEREDASj is to provide simpler descriptions (in contrast to OWL-S [153] or WSMO [141]) that contain necessary semantics (such as SA-REST and MicroWSMO, see Sect. 5.2.2) in a widely used machine-targeted media type (*application/json*). SEREDASj defines a syntactic structure for JSON documents and a corresponding interpretation. It enables the representation of hypermedia links, which are not natively present in JSON. It is, however, not a strictly validated approach such as JSON schema [258], which enforces the presence of certain elements and properties in a JSON document.

The authors of SEREDASj put forward three use cases [137]. Firstly, they envision it as a means of creating documentation, both in machine-processable and human-readable form. The human-readable counterpart is generated from labels of predicates defined in ontologies, which are referenced by URI in the SEREDASj description. The benefit here is reuse—on the one hand by having a single description for humans and machines, and on the other hand by referencing to existing ontological definitions. Secondly, the goal is to enable more flexibility in Web API clients. By adding support for hyperlinks to JSON, SEREDASj descriptions become a hypermedia format through which clients can navigate a Web API in accordance with the hypermedia constraint [78]. Thirdly, SEREDASj aims to facilitate data integration, as its annotations enable the transformation of JSON into RDF. However, as we will argue below, other means to this end exist, so employing SEREDASj specifically for this last use case might prove suboptimal.

Generally speaking, one SEREDASj description is created per resource type. As SEREDASj currently only supports JSON, it is assumed that resources of this type have at least a JSON representation. The accompanying SEREDASj description documents the elements in these JSON documents by mapping them to predicates of ontologies in RDF format. This principle can be applied to the whole hierarchy of the document, including arrays, which represent multi-valued properties. Every subtree can additionally be associated with an RDF type. Next to this, SEREDASj describes the controls to navigate through the Web API in the form of URI templates [99], as well as the RDF predicates they correspond to, capturing their meaning. SEREDASj can furthermore detail the format of entities for use in PUT or POST requests.

Part of the functionality of SEREDASj is currently offered by JSON-LD [139], whose specification is currently a W3C editor's draft [218]. JSON-LD similarly provides predicate and type annotations that allow JSON data to be translated into RDF, but these annotations are included in the JSON document itself as opposed to a separate SEREDASj description document. However, the question arises whether it would not be more beneficial for servers to provide separate JSON and RDF representations of a resource, allowing clients to indicate through content negotiation with which representation they would like to proceed.

Further work on SEREDASj includes an architecture to integrate Web APIs into the Web of Data, which makes use of SEREDASj descriptions [140].

5.6 Tools

In previous sections, different solutions for describing semantics of REST APIs have been investigated. However, there are some obstacles preventing them from being widely adopted. First, writing semantic service descriptions by hand is a time consuming and tedious task. Furthermore, to model APIs, most of these approaches require some degree of expertise in Semantic Web languages such as RDF, SPARQL, and N3 in addition to the domain knowledge. Tools can play a significant role by providing a user interface to rapidly build semantic descriptions, making the complexity of formal specification transparent to the user.

The here introduced formalisms, including MicroWSMO, SA-REST and the MSM, which make HTML service descriptions machine-processable and enable the adding of semantic information, provide the means for creating semantic descriptions of Web APIs. However, without supporting tools or guidelines, developers would have to modify and enhance the descriptions manually by using a simple text/HTML editor. In addition, the complete annotation process would have to be completed manually, if there are no tools, which enable the search for suitable domain ontologies or the reuse of annotations of previously semantically described services.

5.6.1 Karma

Karma[13] [232] is a Web-based framework for integrating structured data from a variety of sources. Users can load data from relational databases, spreadsheets, delimited text files, *kml* (Keyhole Markup Language) files, and semi-structured Web pages. Users can clean and normalize data with a programming by example interface [255]. Then, Karma semi-automatically builds a semantic model of the source by mapping it a domain ontology chosen by the user [124]. Karma models each column in terms of the classes and data properties defined in the ontology, and models the relationships among columns using object properties. Once data is modeled, Karma can translate the data into a variety of formats including RDF. The semantic models also enable Karma to integrate information from multiple sources and to invoke services to compute new information.

The main goal in Karma is to make it easy and efficient for users to perform all information integration tasks. Karma enables users to perform operations on a small set of input instances, and then learns from these examples a general procedure that it can apply to all inputs. Compared to other data integration tools, Karma significantly reduces the time and effort needed to perform the data integration tasks.

Recently, Karma has been extended with the capability to build semantic descriptions of Web APIs as a foundation to compose data sources and Web services [223].

[13] http://www.isi.edu/integration/karma

In this section, we explain how it enables users to rapidly generate semantic service descriptions.

To model services, Karma first asks the user to provide samples of the API invocations URLs. This conforms to the main idea of Karma that examples are the basis to carry out the tasks. These sample URLs can also be automatically extracted from the documentation pages of the APIs. Next, the user interactively builds a semantic model of the API by mapping the service inputs and outputs to the domain ontology. Building semantic models is the central part of the approach and it is very similar to how Karma models the data from other sources. Once the semantic model is built, Karma formalizes it using a new expressive RDF vocabulary that represents both the syntactic part and the functionality of the API. Karma stores the service specifications in a triple store, enabling users to query the service models using SPARQL. Finally, Karma deploys a Linked API that consumes and produces Linked Data. The Linked API provides REST interfaces enabling users to send RDF data in the body of a POST request and get back RDF output linked to the input. In the following paragraphs, we explain each step in more detail.

The input to the system is a set of examples of the API requests. Karma parses the URLs and extracts the individual input parameters along with their values. For each invocation example, Karma calls the API and extracts the output attributes and their values from the XML or JSON response. Then, Karma joins the input and the output values into one table and shows that in a worksheet. Karma treats this table as a regular data source and applies its source modeling technique to build a semantic model of the API.

The goal of semantic modeling is to express the API functionality in terms of classes and properties defined in a domain ontology. The modeling process consists of two steps. The first step is to identify the type of data by assigning a *semantic type* to each column. A semantic type can be either an ontology class or a pair consisting of a data property and its domain. Karma uses a conditional random field (CRF) [135] model to learn the assignment of semantic types to columns of data [93]. Karma uses this classifier to automatically suggest semantic types for new data columns. If the correct label is not among the suggested labels, users can browse the ontology through a user-friendly interface to find the appropriate type. The system re-trains the CRF model after these manual assignments.

The second part of the modeling process is to extract the relationships between the inferred semantic types. Given the domain ontology and the assigned semantic types, Karma creates a graph that defines the space of all possible mappings between the source and the ontology [124]. The nodes in this graph represent classes in the ontology connected by direct and inferred properties. Once Karma constructs the graph, it computes the API model as the minimal tree that connects all the semantic types. It is possible that multiple minimal trees exist, or that the correct model of the data is captured by a non-minimal tree. In these cases, Karma allows the user to interactively impose constraints on the algorithm to build the correct model. Karma provides an easy-to-use *gui* where the user can adjust the relationships between the columns.

The models that Karma builds are themselves represented in RDF according to an ontology[14] reusing existing ontologies such as SWRL[15] and hRESTS[16] [223]. This ontology is semantically richer than WSMO-Lite[17] and Minimal Service Model (MSM) [187] because in addition to annotating each input and output with semantic types, it also explicitly represents the relationships among inputs and outputs. Another advantage of the Karma models is that they are represented in RDF, making it possible for clients to query and discover models using SPARQL. Other approaches use graph patterns to represent the service models, so it is not possible to use SPARQL to query the models.

The Karma API models are expressive enough that it would be possible to export then to other formal specifications such as LOS and MSM. This is a direction for future work [224].

Karma also has a Web server where the modeled API will be deployed as a Linked API. The Linked API implements a REST interface allowing clients to send RDF data in a POST request. One benefit of service descriptions in Karma is that they contain all the information needed to automatically execute APIs and do the required lowering and lifting, obviating the need to manually write separate instructions using formalisms such as XSLT and SPARQL. Once the Web server receives the user POST request, it uses the service description in the Linked API repository to lower the RDF data and build the invocation URL. Then, it invokes the Web API and again uses the service description to automatically lift the XML or JSON response to generate linked data. Karma is available as an open source[18] software and users can use it to model the APIs based on their own needs.

5.6.2 SWEET

To facilitate the easier adoption of semantic description of Web APIs by supporting users in their creation, KMi has developed SWEET: Semantic Web sErvices Editing Tool[19].

SWEET is developed as a Web application that can be launched in a common Web browser and does not require any installation or additional configuration. It provides key functionalities for modifying the HTML Web API descriptions in order to include markup that identifies the different parts of the API, such as operations, inputs and outputs, and also supports the adding of semantic annotations by linking the different service parts to semantic entities. As a result, SWEET enables the creation of complete semantic Web API descriptions, based on the previously introduced models, given only the existing HTML documentation. More importantly, the

[14] http://isi.edu/integration/karma/ontologies/model/current#
[15] Semantic Web Rule Language: http://www.w3.org/Submission/SWRL
[16] http://purl.org/hrests/current#
[17] http://www.w3.org/Submission/WSMO-Lite
[18] http://github.com/InformationIntegrationGroup/Web-Karma-Public
[19] http://sweet.kmi.open.ac.uk

tool hides formalism and annotation complexities from the user by simply visualizing and highlighting the parts of the API that are already annotated and produces an HTML description that is visually equivalent to the original one but is enhanced with metadata that captures the syntactical and semantic details of the APIs. The resulting HTML description also serves as the basis for extracting an RDF-based semantic Web API description, which can be published and shared in a service repository, such as iServe, enabling service browsing and search.

SWEET[20] is a Web application developed using JavaScript and Ext*gwt*[21], which is started in a Web browser by calling the host URL. It is part of a fully-fledged framework, developed within the scope of the SOA4All European project[22], for supporting the lifecycle of services, particularly targeted at enabling the creation of semantic Web API descriptions. SWEET takes as input an HTML Web page documenting a Web API and offers functionalities for annotating service properties and for associating semantic information with them. A current version of the tool can be found online[23].

SWEET is designed as a classical three-layered Web application. The architecture of SWEET consists of three main components, including the visualization component, the data preprocessing component and the annotations recommender. The visualization component is based on a model-view-controller architecture design pattern, where the model implements an internal representation of the annotated Web API, in accordance with the elements foreseen by the semantic formalisms detailed in the previous sections. In this way, every time the user adds a new annotation via the interface, the model representation of the Web API description is automatically updated. Similarly, when parts of the model representation are altered or deleted, the highlighting and visualization in the user interface is also adjusted. When the annotation process is complete, the resulting HTML and RDF Web API descriptions are generated based on the produced internal model.

The GUI of the visualization component is shown in Fig. 5.3 and it has three main panels. The HTML of the Web API is loaded in the *Navigator* panel, which implements a reverse proxy that enables the communication between the annotation functions and the HTML by rerouting all sources and connections from the original HTML through the Web application. Based on this, the HTML *dom* of the RESTful service can freely be manipulated by using functionalities of the *Annotation Editor* panel. The current status of the annotation is visualized in the form of a tree structure in the *Semantic Description* panel, which is implemented by automatically synchronizing the visualization of the service annotation with an internal model representation, every time the user manipulates it.

In addition to these three main panels, SWEET offers a number of supplementary useful functionalities. It guides the user thorough the process of marking service properties with hRESTS tags, by limiting the available tags depending on the current

[20] http://sweet.kmi.open.ac.uk
[21] http://extjs.com/products/gxt/
[22] SOA4AllEU project FP7 - 215219, http://soa4all.eu/
[23] http://sweetdemo.kmi.open.ac.uk/soa4all/MicroWSMOeditor.html

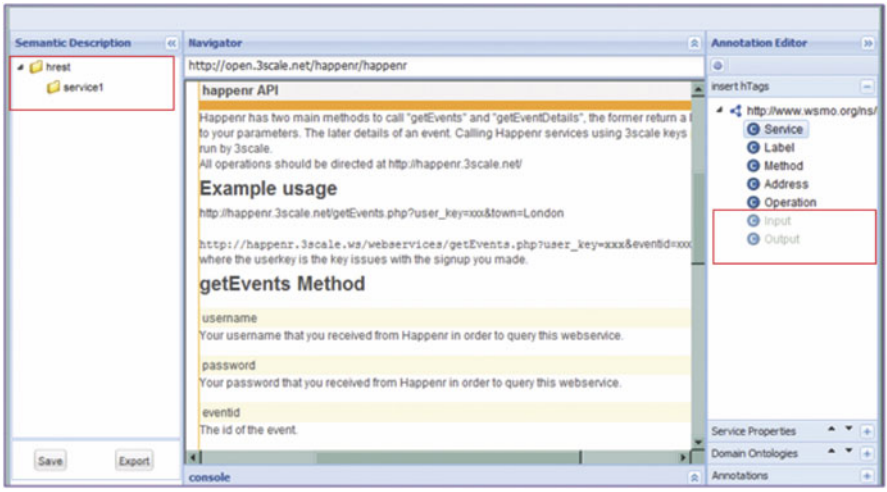


Fig. 5.3 SWEET: Inserting hRESTS tags

state of the annotation. This implements measures for reducing possible mistakes during the creation of annotations. In addition, based on the hRESTS tagged HTML, which provides the structure of the Web API, the user can link service properties to semantic content. This is done by selecting service properties, searching for suitable domain ontologies by accessing Watson [59] in an integrated way, and by browsing ontology information. Based on this details the user can decide to associate a service property with particular semantic information by inserting a SA-WSDL model reference tag.

In summary, SWEET takes as input the HTML Website description of the Web API, which is loaded in the central panel, and returns a semantically annotated version of the HTML or a RDF semantic description. In order to do this the user needs to complete the following four mains steps:

1. Identify service properties (service, operation, address, HTTP method, input, output and label) by inserting hRESTS tags in the HTML service description.
2. Search for domain ontologies suitable for annotating the service properties.
3. Annotate service properties with semantic information.
4. Save or export the annotated Web API.

The first step can easily be completed by simply selecting the part of the HTML, which describes a particular service property, and clicking on the corresponding tag in the *inset hTags* pane. In the beginning, only the *Service* node of the hRESTS tree is enabled. After the user marks the body of the service, additional tags, such as the *Operation* and *Method*, are enabled. In this way, the user is guided though the process of structuring the RESTful service description and is prevented from making annotation mistakes. After the user structures the HTML description and identifies all service properties, the adding of semantic information can begin. The new version

of SWEET, just like the bookmarklet, supports users in searching for suitable domain ontologies by providing an integrated search with Watson [59]. The search is done by selecting a service property and sending it as a search request to Watson. The result is a set of ontology entities, matching the service property search. Once the user has decided, which ontology to use for the service property annotation, he/she can do an annotation by selecting a part of the service HTML description and clicking on *Semantic Annotation* in the *Service Properties* context menu. This results in inserting a model attribute and a reference pointing to the URI, of the linked semantic concept.

The resulting descriptions can be directly posted to iServe [186] or can be re-posted on the Web. The use of the microformat tags enables the automated search and crawling for APIs, since it serves as a basis for distinguishing simple HTML websites from Web API descriptions. Furthermore, when posted to iServe, the Web API descriptions can be browsed and searched alongside with WSDL-based services. Since the semantic Web API descriptions use SA-WSDL-like annotations in combination with the WSMO-Lite service ontology, both WSDL-based services and APIs can be retrieved by using the same queries. For example, a search query for music services would return the Last.fm description as well as all other APIs or services related to music. As a result, all type of services, can be retrieved in a unified way.

5.7 Open Problems and Future Work

5.7.1 Cross-Origin Resource Sharing (CORS)

The XMLHttpRequest specification [19] defines an API that provides scripted client functionality for transferring data between a client and a server. In today's common Web applications like online spreadsheets, word processors, presentation tools *etc.*—and even more in so-called mash-up applications[24]—the majority of data transfers between server and client happen based on XMLHttpRequest. An important security aspect of XMLHttpRequest, however, is the so-called Same Origin Policy (SOP). This policy permits scripts running on pages originating from the *same* site to access each other's methods and properties with *no* specific restrictions, however, *prevents* access to most methods and properties across pages on *different* sites. While providing at least some protection from rogue Web pages accessing private data, SOP also has severe implications for cases where cross-origin data transfers are actually legit. Past attempts to legally circumvent SOP include using proxy servers, Adobe Flash, and JSON-P,[25] however, more recently, the tendency goes in the direction of properly handling cross-origin resource sharing (CORS) through a mechanism documented in a em w_3c Working Draft [236]. The CORS standard works by adding new HTTP

[24] Web applications that combine data from multiple sources to create new services, many of them listed in [64].

[25] First documented appearance of JSON-P in Bob Ippolito's December 2005 blog post: http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/.

headers that allow servers to serve resources to permitted origin domains. Browsers support these headers and enforce the restrictions they establish. While not all APIs support CORS yet, there is a remarkable momentum of Web API and data providers in general to open up their data and becoming CORS-enabled[26].

5.7.2 Authentication

The IEEE defines[27] authentication as *the act of confirming the truth of an attribute of a datum or entity. This might involve confirming the identity of a person [...], or assuring that a computer program is a trusted one*. In the world of APIs, simple authentication paradigms include (but are not limited to) API keys (codes passed in by computer programs calling an API to identify the calling program, its developer, or its user to the API provider), HTTP Basic authentication, HTTP headers, or HTTP cookies. In recent times, different versions of the *authorization* protocol OAuth[28] gain traction as the *de facto* default standard for authorization. *Authentication* is the mechanism whereby systems may securely identify their users. *Authorization*, by contrast, is the mechanism by which a system determines what level of access a particular authenticated user should have to secured resources controlled by the system[29]. In the case of OAuth, if the user grants access to a resource, the application can retrieve the unique identifier for establishing the identity in turn by using the particular API calls, and thus effectively enabling pseudo-authentication using OAuth.

[149] provides as extensive overview of currently used authentication approaches, the required credentials, ways of transmitting the credentials and used authentication mechanisms. The provided solution is based on defining authentication extensions to the MSM defined in Sect. 2.

5.7.3 CORS and Authentication in API Descriptions

To the best of our knowledge, neither authentication nor CORS are covered by the before-mentioned API description formats, the honorable exception being the Web Application Description Language (WADL [104]), where (authentication) HTTP headers can be described for API query parameters. While the implementation status of CORS can be determined at runtime by examining a sample API request and checking for the existence of the particular HTTP header, there is no general way to discover the authentication requirements of an API at runtime. In consequence, both CORS and authentication and ways to semantically describe them remain a field for future research.

[26] http://enable-cors.org/
[27] http://technav.ieee.org/tag/2585/authentication
[28] http://oauth.net/
[29] http://www.duke.edu/∼rob/kerberos/authvauth.html

5.8 Conclusion

We have surveyed the current state-of-the-art in descriptions of Web APIs and classified the various approaches. The main strength of RESTful APIS, the flexibility which with the APIs can be designed and deployed, at the same time burdens client application developers with the manual work of understanding, interpreting, and reconciling the various approaches to API design. Almost all of today's Web APIs come with a textual description, lacking coherence. A little structure in architecting and documenting the APIs could greatly benefit application developers and reduce the amount of manual effort required when integrating multiple APIs.

Acknowledgments R. Verborgh and R. Van de Walle are funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (iMinds), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union. A. Harth and S. Speiser acknowledge the support of the European Commission's Seventh Framework Programme FP7/2007-2013 (PlanetData, Grant 257641). S. Stadtmüller has been supported by a Software Campus grant.

Chapter 6
APIs to Affordances: A New Paradigm for Services on the Web

Mike Amundsen

6.1 Background

In the last several years, the landscape of the Internet has changed noticeably. There are many more connected devices, more connected applications, and thousands of Web "APIs" to service them. This represents a new "ecosystem" for the Web; one dominated by small devices loaded with specialized applications, all talking across the Web using shared application programming interfaces (APIs). While the shift did not happen all at once, probably the date that best marks the start of this new era in the Web would be January 10, 2007; the day the first Apple iPhone was introduced [100].

The resulting sales boom launched competitors and an industry has grown up around the devices themselves. As an example, even the work force needed to support the creation of applications for hand-held devices is considered worthy of scrutiny.

6.1.1 More Devices

The common wisdom is that the number of devices connected to the Internet is growing rapidly. In May of 2009, Intel predicted that the number of 'connected devices' would grow from the then estimated 5 billion to 15 billion by the year 2015 [60]. In an April 2010 press release [242], the CEO of the Swedish telecommunication company, Ericsson, predicted the number of connected devices will balloon to as many as 50 billion by the year 2020.

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

M. Amundsen (✉)
Principal API Architect for Layer 7 Technologies, Washington, USA,
e-mail: mamundsen@layer7tech.com

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_6,

6.1.2 The "App Economy"

Much has also been made of the "App Economy" —a burgeoning market that exists to feed the many devices we all own and use. This economy is now viewed by some as a "job creator" worthy of special attention. Researcher Dr. Michael Mandel estimates 466,000 new jobs have been created since the introduction of the iPhone in 2007. [151]

These apps are increasingly written in code that is "native" to the device; thus requiring custom builds for each targeted device or family of devices. Not surprisingly, new products have appeared to decrease the effort needed to produce the numerous native code builds (i.e. PhoneGap[1], Appcelerator[2], and others). Of course, there continues to be debate within the mobile community on whether a "native" or "browser-based" approach is preferable.

6.1.3 More Services

To match the growth in devices and applications, a similar growth is occurring in services (or "APIs") to support the ecosystem. The Programmable Web, a site which catalogs such things, recorded more than 8000 Web APIs as of November 2012 and proudly stated "The whole web as a platform has come a long way and done so very quickly." [237]

6.1.4 Summary

Each of these areas of growth represent great opportunities. Growth means new markets, expanding use of the Web, and the chance to improve the quality and quantity of data available to all users.

But there are challenges, too. These rapidly expanding markets are, for the present, continuing to rely on dated technologies and mental models. Despite the incredible growth of the Web both in real (device) and virtual (applications & APIs) terms, the most common approaches to service this market are still based on software theories developed at the dawn of personal computing; when devices where most often "stand-alone" machines, sometimes connected to each other over local area networks, and only on occasion (and for brief periods) connected over slow communication lines to other, remote devices.

[1] http://phonegap.com/
[2] http://www.appcelerator.com/

6.2 The Problem

There are a number of problems posed by the current trend in adding many small, dedicated devices to the Internet. This chapter focuses on a set of problems at the level of software architecture; problems related directly to the work of enabling communication between connected devices.

6.2.1 Technical Difficulties

One type of shortcoming in the common implementation model for Web APIs are technical in nature. Primarily these are due to incomplete or inappropriate implementation of the HTTP protocol. While there are many possible minor difficulties to implementing HTTP (misuse of methods, status codes, headers, etc.) two general problems are identified here:

- Treating HTTP as a Transport
- Lack of Component-Connector Modeling

6.2.1.1 Treating HTTP as a Transport

Most of the Web API implementations continue to use traditional RPC-style interfaces. The messages sent over the wire are simple data blocks, often meant to faithfully represent a server's internal objects or data graphs. Usually the information in carried via a simple data format such as XML, JSON, CSV, etc. This approach has the effect of treating HTTP as a transport protocol [165] and weakens one of the three pillars of Fielding's architectural style: the Data element [76].

When this happens, the ability to communicate options to the recipient is lost. Client applications are expected to have a complete understanding of not only the message received, but also any other possible messages that could be received or even requested. This dependence on clients to make all the decisions means that there is almost no "shared understanding" between clients and servers other than an understanding of the data format used to pass messages back and forth and the protocol used to send them.

Using HTTP as a transport can also result in implementation models that limit client server interactions to only those that easily map to the protocol's method set. This is commonly referred to as HTTP-CRUD [233]. Combining this limitation of the client-server interaction with the dependence on simple data formats for messages not only misses key properties of HTTP, it also limits the usability of the Web for anything other than purpose-built client applications that rely on a static, isolated understanding of the problem domain represented by a single server.

6.2.1.2 Loss of Connector-Component Model

The Connector-Component Model was first described by Taylor, et al in 1995 [226] as Chiron-2 or C2. The original model was used to coordinate independent user interface components using a connector to pass messages between them. Later, Fielding used similar terminology to describe network-level communications where independent components (running on separate machines) use protocol connectors to communicate across the network [76]. For Fielding, Component, Connector, and (as seen above) Data were the three key architectural elements of consideration for his REST style.

This architectural model (one that relies on independent components using connectors as intermediaries) has allowed the Web to continue to scale not just at the runtime level, but also at the implementation level. Since components only need to know how to talk to connectors and connectors only need to know how to talk to other connectors, it is possible to independently build parts of the network with minimal design-time co-ordination between parties.

The recent trend in building "native" applications for connected devices can result in a loss of the component-connector paradigm and, in turn, cause problems at scale; at both the implementation and runtime levels. Employing cross-platform build tools only masks the problem.

6.2.2 Competing Priorities

On a more practical level, creating Web services that are both flexible and easy to use is a challenge. An easy-to-use Web service is essential for attracting customers. A flexible, evolvable service that does not tax developers by obsoleting their work too often, is important for retaining customers over time. In some cases these are competing priorities. For example, flexible evolvable systems can be viewed by developers as difficult to understand.

6.2.2.1 Immediate Usability

As more Web services appear, they compete for the attention of developers and subscribers. Along with reliability and performance, Web service providers are growing increasing aware that usability is a key factor for increasing adoption.

If developers cannot understand the Web service, can't easily connect and quickly build working solutions for it, they are likely to look elsewhere. In addition, once a developer settles on a provider, that developer is not likely to switch to a new provider since most web service integration today requires special coding for each and every service; even for the same service provided by two different vendors (i.e. peer-to-peer messaging, photo-sharing, etc.).

Thus usability is a key factor in acquiring new customers. Web services need to offer familiar, easy-to-understand examples and documentation in formats developers understand and can quickly convert into working code.

6.2.2.2 Long-Term Evolvability

At the same time, one of the desirable properties of large-scale systems is the ability to support evolution of the service over time; evolvability. Fielding defines this as "Evolvability represents the degree to which a component implementation can be changed without negatively impacting other components." [76]

A common way to support long-term evolvability is to use hypermedia affordances within response representation. Fielding's REST style, which relies on the use of hypermedia was conceived with long-term evolvability in mind. In a 2010 interview Fielding states "Most of REST's constraints are focused on preserving independent evolvability over time, which is only measurable on the scale of years." [161]

6.2.3 The Time Dimension

Another important aspect of supporting large networks is the element of time; not at the micro, but the macro level. Few architectural models consider the passage of time and how it affects both the participants and the messages they share.

6.2.3.1 REST Resources Over Time

In Fielding's REST style, a resource (that which can be named) is "a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time." In this way, not only the *representation* of the resource can change over time, the data upon which the resource is based may change as well; including a case that results in an "empty" resource. Under these conditions, enforcing contracts on *what an identifier will return* at some point in time in the future is likely impossible. Instead, in REST, "The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another." In other words, the conceptual meaning identified when the resource was created is expected to remain constant. If, at the time of creation, the identified resource is meant to represent "the latest edition of Document A", it can be expected to continue to represent "the latest edition of Document A" at any future point. However, the *contents* of that document cannot be expected to remain static over time.

6.2.3.2 Static Contracts

Unlike Fielding's focus on the semantics of the identified resource, the most common attempts to describe how clients and servers interact over the network (i.e. WSDL [50], WADL [103], etc.) rely on static interface contracts that do not change over time. Whereas message-based media types whose processing models describe possible affordances (controls) which could appear within responses, typical Web API contracts contain a set of function calls (including arguments and return types) which the client can use to "compose" their own set of interactions with the server (usually after some level of coordination via additional documentation). Web APIs have no clear means for communicating the semantics of the identified resources.

Static contracts can result in 'frozen' implementations that are unable to easily evolve over time as the problem domain changes. Even more important, focusing on 'native' implementation strategies can result in mixing the connector semantics (i.e. HTTP protocol) with the component semantics (i.e. the problem domain) in ways that make it more difficult to modify the application over time.

It is possible that application vendors have little incentive to create long-lived applications since their revenue maybe derived from the constant update/replacement of 'obsolete' apps. However, many app providers may not be focused on gaining revenue through the replacement of what could be perfectly acceptable applications if implemented differently.

6.2.3.3 Transient Devices, Persistent Networks

Current trends indicate that hand-held computing devices are not only seen as essential in today's world, they are also treated as disposable. A 2010 report estimated Americans dispose of 130 million cell phones each year [213].

While devices can be viewed as transient, the networks themselves continue to run 24x7 and, now more then ever, are available via wireless connection to the point where it is possible to remain connected throughout the day, even while traveling. With continued use of CDNs (Content Delivery Networks) and the rise in SaaS (Software as a Service), not only can users stay connected, there is increasing likelihood they can access their personal content at all times.

In this light, continuing to focus efforts on programming each device natively, encoding all the domain knowledge on these devices, may not be the best way to make use of software developers' (and architects') energies. Instead it may make more sense to leverage the network itself; to actually program the network instead of the connected devices. This idea represents not just an adjustment in focus, but also a change in the way network applications are architected and implemented.

6.3 Other Disciplines

Before moving on to a proposed alternate paradigm for modeling communication along the network it may be informative to highlight similarities in observations about perception and communicate from other disciplines. Here five perspectives on the way humans perceive and communicate information are offered for the reader's consideration. The purpose here is to identify a similar thread throughout multiple disciplines; a thread that may be applied to modeling communication on widely distributed networks.

6.3.1 Architecture

In 1979 Christopher Alexander released the first in a series of texts describing his approach to physical architecture: "The Timeless Way of Building" [8]. In it, he asserts that "[P]eople can shape buildings ... using *pattern languages*" and that "A pattern language gives each person who uses it the power to create an infinite variety of new and unique buildings, *just as ordinary language gives him the power to create an infinite variety of sentences*."

In 1987, Beck and Cunningham presented a workshop at OOPSLA-87: "Using Pattern Languages for Object-Oriented Programs" [22]. The abstract contained the following report: "Our initial success using a pattern language for user interface design has left us quite enthusiastic about the possibilities for computer users designing and programming their own applications."

For Alexander, patterns (and languages built up from them) are a transcendent means of communication. Alexander's pattern language relies on the notion that all individuals can recognize abstract patterns, regardless of variance of time (over the centuries) or place (Rome, Africa, China, the Americas, etc.). Beck and Cunningham, and many who followed them, were able to apply this same notion to software designed to support direct human interaction (i.e. graphical interfaces).

6.3.2 Visual Perception

Around the time that Beck and Cunningham were applying Alexander's pattern model to implementing graphical user interfaces, psychologist James Gibson explored the concept of "affordance" in his book "The Ecological Approach to Visual Perception." [92] For Gibson, affordances were the "action possibilities" of the environment in which the subject resides. These possibilities were perceived by the subject in relation their abilities. For example, an short opening two feet wide might be perceived as a "doorway" to a small creature, but not to an animal six feet tall.

Gibson claimed that animals continually "sampled" their surroundings and made decisions based on the affordances available to them at the moment. For Gibson, the

world was divisible into ecological niches—each with their own set of affordances—and animals within that niche became expert at exploiting the available affordances. Affordances (and therefore options) were everywhere and, like Alexander, Gibson believed these affordances could be described in general terms (doorway, chair, step, etc.) that applied across environments.

6.3.3 Industrial Design

The notion of affordances was further refined by Donald Norman in his 1988 book "The Design of Everyday Things." [167] Norman applied Gibson's ideas to industrial design and HCI (Human-Computer Interaction) to help launch the field of Usability. Along the way Norman identified the Seven Stages of Action to describe how humans usually interact with their environment in order to accomplish a goal:

1. Set a goal
2. Form an intention to reach that goal
3. Specify and action
4. Execute that action
5. Perceive the state of the world
6. Interpret the state of the world
7. Evaluate the outcome against the goal.

Norman also described the notion that humans approach the environment (and its affordances) with some level of information already "in the head" and use their perception to discover information "in the world". It is this mix of "in the head" and "in the world" that determines the usability of an object (for that individual).

By detailing a series of steps humans use to reach their goals and acknowledging that information resides both within and without the individual, Norman's vision of the world includes not just Alexander's patterns and Gibson's affordances, but also the knowledge and goals of the participant.

6.3.4 Cross-Cultural Mono-Myth

American mythologist and writer Joseph Campbell described a different kind of shared pattern in his 1949 work "The Hero with a Thousand Faces." [42] For Campbell the "hero's journey" was a story which not only appeared in multiple cultures across both space and time, but it retained the same general pattern which he summarized as follows:

> A hero ventures forth from the world of common day into a region of supernatural wonder: fabulous forces are there encountered and a decisive victory is won: the hero comes back from this mysterious adventure with the power to bestow boons on his fellow man.

Campbell's work explored the idea that this shared story was evidence of communication based on archetype and metaphor; something that transcends any single language or culture. For Campbell, the Mono-Myth was a way to share understanding across the divides of clan, kingdom, and time. However, unlike Alexander who focused on self-standing patterns in the world, Campbell asserts that entire portions of culture and story can be viewed as a single "shared understanding."

6.3.5 The Map is not the Territory

Another view of human communication was put forward by Alfred Korzybski in his 1933 tome "Science and Sanity." [130] Here, Korzybski outlines his view that human knowledge is limited not only by our ability to perceive the world but also the language we use to describe what we perceive. For him, our perception is always incomplete; always missing details and filtered by our current beliefs. It was Korzybski who coined the phrase: "The map is not the territory."

Korzybski acknowledged that this ability to function using only a general description of the world allowed humans to develop language, create names for things and share understanding (however imperfect). Shared understanding of the general nature of the world is how humans successfully interact with the environment and it is through language that they share knowledge over space and time. Thus, like Campbell, Korzybski saw understanding as rooted in the language we used to describe our surroundings.

6.3.6 Summary

What all these examples have in common is the notion that there are identifiable entities (patterns, affordances, general concepts, etc.) to which we all can relate; across culture, time and distance. For Alexander, a "doorway" is a universal concept that can be referenced by all. For Gibson, this "doorway" is recognizable even when it has only the barest visual resemblance to our common idea of a doorway (i.e. an entrance to a cave). For Norman, doorways can be rendered more (or less) usable through the application of design principles that can be applied across cultures. For Campbell, the possible meanings of a doorway (as a threshold to a magical place, as a metaphor for moving to a new stage of life, etc.) are also shared.

And finally, Korzybski tells us that all communication is essentially approximate. That the concept of a "doorway" is useful (possibly more so) when it's left vague and general. This generality of shared semantic understanding is the basis for our ability to communicate. We'd be eternally frustrated and lost if we could only successfully communicate when all of us agreed on the exact meaning of every utterance.

So, if the notion of shared understanding through general concepts - ones that are only loosely defined - is a useful paradigm in the fields of architecture, product design, psychology, and story-telling, could it not also be useful in the design and implementation of systems built specifically for sharing information? Do these other disciplines lead us to an alternate way of thinking about information networks themselves?

Instead of working to narrow the scope and meaning—to remove the ambiguity—of network communications; instead of working to create static interfaces that are tied to a specific place and time, maybe there is a way to mimic pattern languages on the network itself; to improve shared understanding by dealing in general concepts communicated through the use of mutually understood affordances in ways that allow for local interpretation and embellishment without the loss of basic meaning.

What would such a system look like? How would it be organized? What are the details of how this kind of communication can be shared over the network?

6.4 An Affordance Paradigm

If we accept the notion that our view of the world is, by nature, always imperfect and that it is shaped not only by our observation but also the language we use to describe it then it may be possible to us a new description language to help us alter our view of the way devices can communication over the Web.

To this end, this chapter identifies two new maxims for the implementation of distributed network applications:

1. Program the network, not the device
2. Rely on affordance-rich messages for communication

The current paradigm for programming network devices is to use the available network as a mere transport over which to ship serialized objects and data graphs based on static programming interfaces. This model is based in the early event-driven, object-oriented paradigm typified by Smalltalk-80. [94] This approach fit well with the (then new) pattern theories of Alexander and Beck & Cunningham. It made sense for handling the interaction of small components arranged within a local graphical interface.

But a model for enabling communication between UI components all running in the same computing space is not the best approach for enabling communication between components over a widely distributed heterogeneous network. Fielding outlined this point of view in his 2001 dissertation [76] using an example architectural style he labeled REST. What Fielding did not fully explore, however, were the details of what data messages in his REST style looked like and how components and connectors worked together to enable communication via these messages. One line from the dissertation has been singled out as the only description of what this message style might entail:

> REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

6.4.1 Affordance-Rich Messages (ARMs)

Fielding mentioned hypermedia as the way to modify state on a distributed network. In this he meant "the simultaneous presentation of information and controls such that the information becomes the affordance through which the user obtains choices and selects actions." [77]

These affordances are, essentially, the pattern language described by Alexander. They are the environmental elements of Gibson's world. Fielding's description also tracks very closely to that of Norman. Applications that rely not just on rules and operations within the code ("in the head" as Norman would say) are capable of recognizing and reacting to affordances in the message itself (similar to Norman's "in the world"). Affordance-rich messages allow for increased shared understanding. They represent more than raw data passed between parties. Additional information about the semantics associated with the data and the options available at that moment in time are also available to the recipient.

And, as outlined by Campbell and Korzybski, sharing the exact meaning of each and every affordance and semantic detail is not necessary in order to share general understanding. Knowledge that an in-message control "affords sending data" or "affords filtering results" is sufficient to support a wide range of operations. Rather than designing systems that fail unless all the exact details in are in place, a better approach is to design what Norman refers to as "explorable" systems; ones that allow users to safely make attempts at reaching their goals.

> Think of each action by the user as an attempt to step in the right direction; an error is simply an action that is incompletely or improperly specified. . . . Try to support, not fight, the user's responses. . . . Design explorable systems. [167]

Affordance-Rich Messages (ARMs) can make it possible to design "explorable" systems.

6.4.2 Programming the Network

Programming the network means focusing on the messages that are passed along the network instead of the devices sending and receiving those messages. There are a number of reasons this shift from device-orientation to network-work orientation can benefit software architects and developers.

First, in widely distributed networks where components have little to no view into the workings of other components on the network, the message passed between them is the only means "in the world" by which communication is possible. In this light,

it makes sense to consider a way in which the network itself can be "programmed." It is through messages that both sender and recipient share understanding and the network is where this understanding lives.

Second, programming the network is not only possible, it is essential in order to support networks as they grow in both the space and time dimension. Think of the possibility of communicating with devices at great distances (i.e. in outer space). It is reasonable to include affordances and additional options within messages that might not reach their recipient for minutes, hours, or even days in order to allow the recipient to engage in additional evaluation of available options before making a (local) decision. As the reach of the network grows, the messages carried by that network need to be more rich and informative.

Finally, by adopting a paradigm where the messages contain "programming code", the network of machines that touch the message as it moves along to its final destination can participate in the communication. The HTTP protocol today is designed to support one level of communication at the network level (the HTTP Header space) and one at the sender-recipient level (the body). Manipulating the message metadata sends signals to intermediaries along the network path; signals devices may understand and act upon when appropriate. This, too, is programming the network.

The network is "programmable" today using ARMs and message metadata over HTTP.

6.4.3 A Working Model

What would a working model of a programmable network look like? We already have all the technical tools needed to make this a reality. What is needed is to delineate the necessary parts of a working system and show how they can be used to support the proposed paradigm.

The following elements of a working model for programmable networks are described:

- ARM-Aware Network
- ARM-Capable Devices
- ARM Design Model
- ARM Evaluation Model

NOTE: While this section of the paper describes the working model using HTTP as the protocol, ARM-style communication is protocol-agnostic as it sits atop the transfer protocol. As HTTP changes and/or new transfer protocols become available, the ARM paradigm can still be a viable model for programming the network.

6.4.3.1 An ARM Network

ARMs are of benefit when network communication may span notable distances. When these distances (either in space or time) are long enough to be noticeable

by either the recipient or sender (i.e. messages take more than a few seconds to travel between parties), enriching the message with affordances that explain what the recipient can do adds benefit. This includes allowing recipients to store (and possibly forward) messages for later use.

HTTP request/response today has all the properties needed to support this aspect of programmable networks. The HTTP header space contains enough information to know whether the message is fresh or stale, the origin of the message, etc. HTTP responses may also include a body (the part that holds the ARM) which recipients can parse, process, and act upon independent of the sender who originated the message.

A network of machines that understands HTTP can be an ARM-Aware network.

6.4.3.2 ARM-Capable Devices

Currently most connected devices utilize either a generic HTTP browser or a "native" application with a built-in HTTP library in order to communicate along the network. ARM-Capable devices would be able to leverage available support for a protocol (i.e. HTTP) plus one or more ARM processors. This is close to the way Web browsers work; they have strong support for the HTTP protocol and a limited number of affordance-rich media types (i.e. HTML). However current browsers do not easily support adding new media-type processors. ARM-capable devices would be able to treat ARM processors as 'plug-ins' and make it easy to upgrade a device by adding new ARM processors.

ARM-capable devices are ones that not only have strong support for one ore more network protocols, they also have support for multiple message models and/or can add new message models as they become available.

6.4.3.3 ARM Design Model

In an environment where networks support ARM-style communication and devices can support new ARM processors as they become available, having a clear design model for affordance-rich messages is critical. Luckily, one already exists: Hypetext media types or Hypermedia Types. [14]

```
<root>
  <customer id="123">
    <name>Smith,Inc.</name>
    <region>South</region>
    <balance>1000</balance>
  </customer>
</root>
```

Example 1: Non-ARM Response

Media types with native hypermedia controls provide the affordances needed to support ARM-style designs (compare Examples 1 & 2). The author has previously identified a candidate set of these affordances as H-Factors in the book “REST: From Research to Practice.” [14] Additional material on a design methodology for Hypermedia Types was detailed in “Building Hypermedia APIs with HTML5 and Node.” [12]

```
<root>
  <customer id="123" rel="item" href="...">
    <name>Smith,Inc.</name>
    <region>South</region>
    <balance>1000</balance>
    <link rel="edit" href="..."/>
    <link rel="collection" href="..."/>
    <link rel="search" href="..."/>
  </customer>
</root>
```

Example 2: ARM-style Response

Designing messages that carry affordances is, essentially, designing a language through which clients and servers can share understanding. Designs can be targeted (See Example 2) or very general (See Example 3). The design style chosen should fit the needs of the network participants and the nature of the problem domain.

```
<root>
  <item id="123" rel="customer" href="...">
    <dataname="name">Smith,Inc.</data>
    <dataname="region">South</data>
    <dataname="balance">1000</data>
    <link rel="edit" href="..."/>
    <link rel="collection" href="..."/>
    <link rel="search" href="..."/>
  </item>
</root>
```

Example 3: General ARM-style Response

6.4.3.4 ARM Evaluation Model

Producing affordance-rich messages is only helpful if network participants can “understand” and use them when they arrive. A consistent evaluation model for ARMs is needed; one which clients and servers can count on when attempting to use ARM-style responses.

Donald Norman's seven stages of action (see above) provides an likely candidate for evaluating (and acting upon) ARM-style responses. Network participants can be coded to perform the same general steps as humans when interacting with the environment:

1. Identify a goal
2. Establish a set of tasks to reach that goal
3. Execute the identified task
4. Capture the results of that action
5. Evaluate the captured results
6. Compare the results to the identified goal

These steps can be applied to response messages on the network by creating ARM processors that can identify goals, establish, execute, and evaluate the results of tasks performed to reach that goal. This is the heart of any goal-seeking state machine. There are quite a number of possible ways to implement ARM processors. It could be done using a human to perform the stages directly or via automation by relying on "crowd-sourcing" logic for the establishment of tasks and the evaluation of the results.

By treating each ARM design separately and providing processors that understand the ARM "language", connected devices can become active participants in the programmable network.

6.5 Related Work

While much of the ideas in this paper have been identified previously, there are no tangible ARM-style implementations extant on the Web today. There are however, some encouraging examples. The items mentioned here fall short of the paradigm of "programming the network through affordance-rich messages" but they do represent attempts to solve the same problem or mitigate similar perceived shortcomings in the current implementation models.

6.5.1 Web Intents

Paul Kinlan's Web Intents [121] "is a discovery mechanism and extremely light-weight RPC system between web apps, modeled after the similarly-named system in Android." While a decidedly RPC-style approach (as opposed to the ARM-style described in this paper), the general aim of Web Intents is similar. Clients are able to discover and register with providers to handle actions using a declarative model from within the common browser. Like the ARM-style paradigm, client applications can "augment" their ability to handle affordances as they appear within responses.

6.5.2 ql.io

Ebay's ql.io[3] project is a "declarative data-retrieval and aggregation gateway for quickly consuming HTTP APIs." Even though this project does not use affordances as means to interact along the network, it *does* provide a similar service to client applications; a gateway to normalize Web API interactions. This service, like ARM-style designs makes it possible for client applications to better utilize remote services on the network.

6.5.3 Hypertext Application Language (HAL)

The HAL media type "is a lean, domain-agnostic hypermedia type in both JSON and XML, and is designed specifically for exposing RESTful hypermedia APIs" and it could be used to craft ARM-style responses. HAL defines a small set of hypermedia affordances (Resources and Links) and allows designers to use Link Relations[171] to add semantic meaning to the representations. In this way, HAL represents a tangible example of a message design aimed at increasing the reliance on affordances used in network communication.

6.6 Conclusion

Moving from an object-oriented API paradigm to a network-oriented affordance paradigm allows software architects and developers to begin programming the network using affordance-rich messages (ARMs) instead of using traditional functional APIs to program the devices connected to the network. This requires a working model where 1) the network is able to support ARM-style responses (which HTTP does today), 2) connected devices understand not just the transfer protocols in use (HTTP, FTP, IRC, etc.) but also the ARMs being transferred, 3) message designs that allow developers to successfully map actions to affordances, and 4) a message evaluation model that follows Norman's seven stages of action.

While the current Web can support this paradigm, there are only faint examples of this model emerging at this time (i.e. Web Intents, ql.io, HAL, JSON-LD). What is needed at this time is an increased focus on coding clients that can support ARM evaluation and a parallel increase of new ARM-style implementations and message designs.

[3] http://ql.io

Chapter 7
Leveraging Linked Data to Build Hypermedia-Driven Web APIs

Markus Lanthaler

7.1 Introduction

The fact that the REST architectural style forms the fundament of the Web, the most successful distributed system of all time, should be evidence enough of its benefits in terms of scalability, maintainability, and evolvability. One of the fundamental principles of REST is the use of hypermedia to convey valid state transitions at runtime instead of agreeing on static contracts at design time. When building traditional Web sites, developers intuitively use hypermedia to guide visitors through their sites. They understand that no visitor is interested in reading documentation that tells them how to handcraft the URLs necessary to access the desired pages. Developers spend considerable time to ensure that the site is fully interlinked so that visitors are able to reach every single page in just a few clicks. To achieve that, links have to be labeled so that users are able to select the link bringing them one step closer to their goal. Often that means that multiple links with different labels but the same target are presented to make sure that a visitor finds the right path. This is most evident when looking at the checkout process of e-commerce sites which usually consists of a single path leading straight to the order confirmation page (plus a typically de-emphasized link back to the homepage or shopping cart). On this path, the user has to fill in a number of forms asking for order details such as the shipping address or the payment details. It is not a coincidence that these forms tend to use exactly the same language on completely different e-commerce sites. It is also not a coincidence that the same names for the form fields are chosen to allow the user's browser to fill the fields automatically in or, at least, offer auto-completion. HTML5 tries to push that even further by introducing an `autocomplete` attribute along with a set of tokens in order to standardize the auto-completion support across browsers[1]. All this

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

[1] http://www.whatwg.org/specs/web-apps/current-work/#autofilling-form-controls:-the-autocomplete-attribute

M. Lanthaler (✉)
Institute for Information Systems and Computer Media,
Graz University of Technology, Rechbauerstraße 12 8010 Graz, Graz, Austria
e-mail: mail@markus-lanthaler.com

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_7,

is part of purposeful optimization with the clear goal to increase conversion rates, i.e., to ensure that visitors achieve their goal. It is therefore surprising to see that most of the time developers completely ignore hypermedia when using the Web for machine-to-machine interaction, or, more commonly speaking, Web APIs.

Instead of using dynamic contracts that are retrieved and analyzed at runtime, which would, just as on the human Web, allow clients to adapt to ad-hoc changes, developers chose to use static contracts. All the knowledge about the API a server exposes is typically directly embedded into the clients. This leads to tightly coupled systems which impede the independent evolution of its components. When a service's *domain application protocol* [179], which defines the set of legal interactions necessary to achieve a specific, application-dependent goal, is defined in a static, non-machine-readable document served out-of-band, it becomes impossible to dynamically communicate changes to clients. Even though such approaches might work in the short term, they are condemned to break in the long term as assumptions about server resources will break as resources evolve over time.

Another problematic aspect in the development of Web APIs is the proliferation of custom data formats instead of building systems on top of standardized media types. This makes it impossible to write generic clients and inhibits serendipitous reuse. As Steve Vinoski argues in his excellent article [243], platforms, although efficient for their target use case, often inhibit reuse and adaptation by creating highly specialized interfaces; even if they stick to industry standards. He argues "the more specific a service interface [is], the less likely it is to be reused, serendipitously or otherwise, because the likelihood that an interface will fit what a client application requires shrinks as the interface's specificity increases." This observation surely applies to most current Web APIs which are, due to their specializations, rarely flexible enough to be used in unanticipated ways.

This chapter is an attempt to address the issues outlined above. We will start off by exploring why hypermedia is not used for machine-to-machine communication and what the current best practices for developing Web APIs are. After introducing Linked Data and JSON-LD, we will present a lightweight vocabulary able to express concepts needed in most Web APIs. Along with a short list of design guidelines, we will finally show how easily all those techniques can be integrated in current Web frameworks. Based on a simple prototype we will then demonstrate how truly RESTful services that are accessible by a generic client can be build in considerably less time by using these technologies. Last but not least, we will give an outlook of how the proposed approach opens the door to other Semantic Web technologies that have been built in more than ten years of research.

7.2 Hypermedia-Driven Web APIs: Challenges and Best Practices

As outlined in the introduction, developers instinctively use hypermedia when building traditional Web sites but seem to ignore it completely when building Web APIs. One of the reasons behind this might be the different level of tooling support. Since,

in contrast to the human Web, no generic clients exist, developers usually not only need to develop the server side part, but also a client (library), which is then used by other developers to access the Web API. It is not surprising that these two components are often tightly coupled, given that they are usually developed in lockstep by the same team or at least the same company.

One of the reasons for this is certainly that for Web APIs no accepted, standardized media type with hypermedia support exists. JSON, which is much easier to parse and has a direct in-memory representation in most programming languages, is typically favored instead of using HTML as on the human Web. Unfortunately, this often leads to the exposure of internals resulting in a tight coupling between the server and its clients. A common example for this is the inclusion of local, internal identifiers in representations instead of including links to other entities. This requires out-of-band knowledge of IRI (Internationalized Resource Identifier) templates to reconstruct the URLs to retrieve representations of entities referenced in such a way. Since, in most cases, the documentation about those IRI templates is not machine-readable, they are hardcoded into clients which consequently means that clients break whenever the server implementation changes.

The current best practice for developing truly RESTful JSON-based Web APIs is to define a custom media type which extends JSON to support labeled hyperlinks. Effectively this means that HTML's anchor or link tags with their relation attributes (`rel`) are imitated by some JSON structure. Since there is a common need for such functionality, there have already been some efforts to standardize such extensions to JSON such as, e.g., Collection+JSON [13], but so far their adoption is very limited. Far more often these proprietary extensions are documented out-of-band on the API publisher's homepage. Apart from the description of how hyperlinks are expressed, these documentations generally also include a list of resource types (such as products and orders) describing their semantics, properties, and their serializations. Last but not least, a number of link relations along with the supported HTTP operations, the expected inputs and outputs, and the consequences of invoking those operations are documented. This allows developers to get an overview of the API's service surface and to implement specialized clients.

Clients for such APIs are usually small libraries to parse retrieved representations, serialize data to be transmitted to the Web API, and to invoke HTTP operations. Not uncommonly, clients for object-oriented programming languages also include classes to represent the various resource types as native objects. While simple libraries are usually statically bound to the server's URL space, more sophisticated clients embrace hypermedia to eliminate this tight coupling. This allows them to browse through the server's information space and, to a certain degree, to dynamically adapt to changes in interaction flows. Typically such client libraries support a static set of resource types and are able to recognize a set of link relations which allows them to find the paths to achieve specific goals. In a sense, it could be said that those libraries *understand* the representations they retrieve from the API, but effectively this understanding is very limited. These clients do nothing more than interpreting representations by looking for specific tokens that are used to trigger certain operations. Since the semantics of those tokens are very weak, no further algorithmic automation is facilitated.

The problem with the practices outlined above is that they result in specialized implementations targeting specific use cases and not generalizations that can be reused across application domains. Therefore, every API created with such an approach is unique and needs to be documented. Even though most of the code to access such services is very similar, there are still minor differences which make it difficult to reuse code and almost impossible to write generic clients. On the human Web this problem is addressed by a generic media type (HTML) which decouples the clients from the servers they are accessing. Admittedly HTML could be used for Web APIs as well, but its nature, which targets human facing Web pages which are essentially graphical user interfaces, is fundamentally different to machine-to-machine communication. A format such as JSON is a much better fit for such use cases which just require the transfer of structured data; having to parse HTML for this has typically a too big overhead. Thus, to solve this problem also for Web APIs, a generic media type with inherent support for hypermedia is needed. Just adding support for hyperlinks to JSON, as most current approaches do, is not enough because it only solves part of the problem. Since the interaction with Web APIs could generally be seen as a data integration problem, other aspects, such as globally unique identifiers for both the entities and their properties, become important as well. As we will see in the next sections, Linked Data with its dereferenceable identifiers along with a serialization format which is 100 % compatible with JSON might be a solution to solve this problem.

7.3 Linked Data and JSON-LD

The human Web consists of billions and billions of interlinked pages. Hyperlinks are such a fundamental building block of the Web's architecture that it feels natural to browse across sites from completely different publishers. It is taken for granted that content links to other relevant content; relevant links are generally seen as a sign of quality. Surprisingly, exactly the opposite is the case for Web APIs. Very rarely do Web services link to external data. As a matter of fact, most times even links to other resources within the same service itself are missing. Therefore, in 2006 Tim Berners-Lee, the inventor of the World Wide Web, postulated the so called Linked Data principles [23] in an attempt to change that. He urged to use URLs to name things that dereference to useful information in a standardized format. Additionally, the returned data should contain links to other relevant data resulting in a giant graph of data. Linked Data could thus be seen as the direct data-centric counterpart of the document-centric human Web. While the amount of Linked Data has grown significantly over the last couple of years, it is still far from mainstream adoption. The reasons for the slow adoption of Linked Data and the Semantic Web technologies are manifold, but the main problem was probably that the community behind them derailed into the artificial intelligence domain instead of concentrating on more practical data-oriented applications. A lot of potential users were alienated by this and developed an aversion to those technologies—a phenomenon we denoted as *Semaphobia* [138]. A solution

to this problem might be a more gradual introduction to those principles and practices by the use of less disruptive technologies. JSON-LD [218] is an attempt to achieve that by providing a generic serialization format for Linked Data on top of the popular JSON.

One of the design goals in the development of JSON-LD was to require as little effort as possible from developers to create and understand JSON-LD documents and thus great efforts were put in its simplicity and terseness. Instead of the normally triple-centric approach that other common serialization formats for Linked Data use, an entity-centric approach was chosen for JSON-LD. The rationale was to resemble the programming models most developers are familiar with and to reflect the way JSON is used. This and the fact that JSON-LD is 100 % compatible with traditional JSON allow developers to build on existing infrastructure investments. Thus, in many respects, JSON-LD forms an entire ecosystem for developers to work with Linked Data without the high entry barrier other Linked Data respectively Semantic Web technology stacks entail.

Additionally to the features JSON provides, JSON-LD supports hyperlinks, universal identifiers for entities and their properties in the form of IRIs, string internationalization, definition and use of arbitrary data types, support for unordered sets and ordered lists, and, last but not least, a facility to express data graphs. These features not only dramatically simplify data integration, which is the underlying problem in most Web API usage scenarios, but also enable developers to express their data with much stronger semantics.

To use JSON-LD's basic functionality, a developer familiar with JSON only needs to know the two JSON-LD keywords `@context` and `@id`. The `@context` keyword is used to include or reference a so called context which allows JSON properties to be mapped to IRIs in order to become uniquely identifiable across the Web and, possibly, dereferenceable. The `@id` keyword does the same for entities by assigning identifiers to JSON objects. It can also be used to express hyperlinks to other resources. A very simple document could look as follows:

```
{
  "@context": "http://example.com/c/person.jsonld",
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/",
  "knows": {
    "@id": "/people/john",
    "name": "John Doe"
  }
}
```

The document above contains information about a person identified by the IRI `http://example.com/people/markus` with the name `Markus Lanthaler`. It also contains a reference to another person whose identifier is `http://example.com/people/john` (the relative IRI is resolved against the document's base IRI). This reference also shows how some of the properties (in this case the name) of a referenced entity can be directly embedded. This allows developers to fine-tune the performance of Web APIs by reducing the number of HTTP

requests necessary for clients to retrieve the desired information. The referenced context maps the JSON properties to IRIs allowing clients to retrieve more information about them. Assuming that FOAF [37] is used as the vocabulary, the context would look something like this:

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": "http://xmlns.com/foaf/0.1/homepage",
    "knows": " http://xmlns.com/foaf/0.1/knows"
  }
}
```

Instead of including a reference to the context, it is also possible to embed it directly into the document. Furthermore, it is possible to include/reference multiple contexts by wrapping them into an array. This allows, e.g., to reference an external context and to overwrite some of the mappings locally in the document.

The alert reader might notice that the document contains a link to the person's homepage (`http://www.markus-lanthaler.com/`) without using the `@id` construct—and also the context does not contain any further information to disambiguate that IRI from a regular string. This is where the `@type` keyword comes into play. It allows type information to be assigned to properties as well as to individual values and entities. The mapping for `homepage` in the context above could therefore be rewritten to include such type information allowing clients to interpret the value as IRI:

```
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    },
```

In the example above, `@id` is used as the value of `@type` to express that the data type is an IRI—all other types are identified, just as everything else, with IRIs. The most commonly used data types are already standardized as part of XML Schema [31] and it is recommended to reuse them whenever possible to improve interoperability. It is also possible to use `@type` directly in a document to express a value's or entity's type. The initial example could for instance be enriched with its type and a typed creation date:

```
{
  "@context": "http://example.com/c/person.jsonld",
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "@type": "http://xmlns.com/foaf/0.1/Person",
  ...
  "created_at": {
    "@value": "2012-09-05",
    "@type": "http://www.w3.org/2001/XMLSchema#date"
  }
}
```

The first use of `@type` above associates a class (FOAF's `Person` class) with the entity identified by the `@id` keyword. The second use of `@type` associates a data type (XML Schema's `date`) with the value expressed using the `@value` keyword. This is similar to object-oriented programming languages where both scalar and structured types use the same typing mechanism, even though scalar types and structured types are inherently different. As a general rule, when `@value` and `@type` are used in the same JSON object, the `@type` keyword is expressing a data type to be used with scalars otherwise it is expressing an entity type, i.e., a class.

Another use of `@value` is to language-tag strings, which is essential for multilingual applications. This can be done with the `@language` keyword which tags a string with the supplied language code. Just as the `@type` keyword it can either be used in the context or, along with `@value`, in a document's body. The example below shows how the academic title of the person could be added in both English and German:

```
{
  "@context": "http://example.com/c/person.jsonld",
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  ...
  "title": [
    { "@value": "MSc", "@language": "en" },
    { "@value": "Dipl.Ing." , "@language": "de" }
  ]
}
```

This brings us to the only case were JSON-LD differs from traditional JSON, i.e., arrays are generally considered as being unordered sets instead of being ordered lists. This stems from the fact that JSON-LD's underlying data model is a set of directed graphs in which edges are inherently unordered. In most cases that is a minor detail that only matters when JSON-LD is transformed to other serialization formats or, e.g., persisted into a database. JSON-LD, however, has also built-in support for ordered lists in the form of the `@list` keyword which can be used to express that an array has to be interpreted as an ordered list. It can either be used directly in the document by wrapping an object with only a `@list` property around the array or be mapped to a property in the context by setting `@container` to `@list` (`@set` can be used to express explicitly that an array has to be interpreted as an unordered set). Both methods are outlined in the following example:

```
{
  "@context": {
    "propertyA": "http://example.com/vocabulary/A",
    "propertyB": {
      "@id": "http://example.com/vocabulary/B",
      "@container": "@list"
    }
  },
  "propertyA": { "@list": [ "a", "b", "c" ] },
  "propertyB": [ "a", "b", "c" ]
}
```

As the examples shown so far already suggest, it is often cumbersome and error-prone to include all IRIs necessary to transform a JSON document to Linked Data. JSON-LD supports two mechanisms to simplify that. The first one is to define prefix mappings in the context to shorten long IRIs. By using prefixes, the context from the beginning of this section could be rewritten to

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "name": "foaf:name",
    "homepage": "foaf:homepage",
    "knows": "foaf:knows"
  }
}
```

This not only makes the context much smaller, but also much more readable, which reduces the cognitive load put on developers. Prefixes can also be used directly in properties in the body of a document. A JSON-LD processor expands all compact IRIs (that is how IRIs using prefixes are called in JSON-LD) by first splitting them into a prefix and a suffix at the colon and then concatenating the IRI mapped to the prefix to the suffix.

Another way to avoid the manual mapping of terms to full IRIs is the use of `@vocab`. It basically defines an implicit global prefix which is used for properties that are not explicitly mapped to an IRI. Since this method automatically affects every non-mapped property in a document, it is recommended to use this mechanism only when (a) every property should be mapped to an IRI and (b) all properties can be mapped to the same vocabulary by using a common IRI prefix. By using `@vocab`, the initial example could be rewritten as follows to automatically expand all properties to the corresponding properties in the FOAF vocabulary:

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/",
    "homepage": { "@type": "@id" }
  },
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/",
  "knows": {
    "@id": "/people/john",
    "name": "John Doe"
  }
}
```

Data serialized with JSON-LD has the form of a graph, and, at times, it becomes necessary to make statements about the graph itself rather than just about the entities, i.e., the nodes, it contains. That is exactly the purpose of the last remaining keyword: `@graph` (actually there is one more, `@index`, which is used for an advanced feature that is beyond the scope of this chapter). It makes it possible to assign properties and an identifier to the graph itself. The following example shows how a graph can be annotated with its creation date:

```
{
  "@context": {
    ...
    "generatedAt": {
     "@id": "http://www.w3.org/ns/prov#generatedAtTime",
     "@type": "xsd:date"
    }
  },
  "@id": "/graphs/1",
  "generatedAt": "2012-09-05",
  "@graph": {
    "@id": "/people/markus",
    "name": "Markus Lanthaler",
    "homepage": "http://www.markus-lanthaler.com/",
    "knows": {
      "@id": "/people/john",
      "name": "John Doe"
    }
  }
}
```

Additionally to the serialization format described above, JSON-LD also defines a number of algorithms and an API [118] to process such documents. The algorithms allow JSON-LD documents to be *expanded*, *compacted*, and *flattened*. Expansion is the process of taking a JSON-LD document and applying all embedded and referenced contexts such that all IRIs, types, and values are expanded in a way that the contexts are no longer necessary. This makes it much easier to write tools and libraries on top of a JSON-LD processor as it already processed all the information contained in the context; flattening simplifies this even more by normalizing a document's structure. The counterpart to expansion is compaction. It takes a JSON-LD document and applies a supplied context such that the most compact form of the document is generated. Furthermore, the algorithms define how JSON-LD can be converted to RDF and vice versa.

7.4 Hydra: A Lightweight Vocabulary for Hypermedia-Driven Web APIs

As shown in the previous section, JSON-LD provides a generic media type for Linked Data, but, to implement a concrete Web API, also a shared vocabulary, understood by both the server exposing the API and the client consuming it, is needed. Hydra, which stands for Hypermedia-Driven API, is an attempt to define a minimal vocabulary to achieve just that. By specifying a number of concepts which are commonly used in Web APIs it can be used as the foundation to build truly RESTful, hypermedia-driven services that can be accessed with a generic API browser. Even though a detailed description of Hydra [136] with all its details is beyond the scope of this chapter, we would like to, at least briefly, introduce the high-level concepts and the rationale behind them.

Simply speaking a RESTful Web API consists of a number of interlinked resources whereby each is identified by an IRI. To find its way through the resource space, a

client has to understand the semantics of a hyperlink, i.e., be able to identify in which relation a resource stands to another resource. Typically those relationships, as well as the resource types themselves, are domain-specific. Nevertheless, it is possible to extract a number of such link relations and resource types that are generic enough to be applicable to a wide range of application domains. Collections are a good example for this. In Web APIs a collection is often used to reference a number of related resources. They are also often used to expose functionality such as the creation of new resources by POSTing representations to the collection's IRI or searching for specific resources in the collection by accessing the collection with specific URL parameters describing the query.

To support these use cases, Hydra defines the resource types `Collection` and `PagedCollection`. `PagedCollection` is a subtype of `Collection` and, instead of holding references to all member resources, just holds a partial list. To navigate through the partial lists of a `PagedCollection`, a client can use its `firstPage`, `nextPage`, `previousPage`, and `lastPage` properties. A `PagedCollection` also has an `itemsPerPage` and a `totalItems` property which tells a client how many items there are in total and how many items are included in a single page. Sometimes it is desirable to allow a client to decide how many items should be returned per page. To achieve that and other functionality, Hydra has built-in support for IRI templates.

An `IriTemplate` has a `template` and a `mappings` property. While the value of the `template` property is simply the IRI template itself, the `mappings` property holds a set of variable-to-property mappings, i.e., every variable in an IRI template can be mapped to a specific property. For instance, in a `PagedCollection` the `firstPage` property can be set to an `IriTemplate` which includes a variable that is mapped to the `itemsPerPage` property. That way a client can control how many items it wants to retrieve per page.

Another common use case is to search for certain items in a collection. Hydra's `Collection` includes a `search` property whose value is an IRI template. A developer can use the generic IRI templating mechanism to add as many search facets as desired. Most of these facets are application-specific and thus not represented in Hydra. The only facet build into it are free-text queries. All a developer has to do to model a free-text query is to map a variable in the IRI template to the `freetextQuery` property.

So far all of the described concepts address read-only aspects of an API, but one of the key differentiators of Hydra compared to other, similar approaches is that it has full read-write support. To facilitate that, every resource might have an `operations` property. This property can hold a set of valid operations that a client might invoke. Each `Operation` consists of an HTTP `method`, an optional `expects` input type, an optional `returns` type on success, as well as a `title` and a `description`. It is also possible to document `statusCodes` that might be returned with a `description`. This allows a developer to understand what to expect when invoking an operation. It has, however, not to be considered as an extensive list of all potentially returned status codes; it is merely a hint. Developers should expect to encounter other HTTP status codes as well. Furthermore, Hydra also has a built-in concept to convey additional information about an `Error` at runtime in case that an HTTP status code is not detailed enough.

To express the semantics of an `Operation` in a machine-readable way, i.e., without relying on its natural-language description, a developer should create specific subclasses thereof. Hydra already contains operations to create, replace, and delete resources out of the box. The `CreateResourceOperation`, e.g., is defined to create a resource of the type specified in `returns` by taking a representation of `expects` as input. One might wonder why the `search` property exists given that everything could be expressed by using operations as well. The reason for this is that the `search` property specifies the relationship between an instance of a `Collection` and a filtered collection which is, strictly speaking, a different resource. Operations, on the other hand, apply to a specific resource or a specific class of resources.

7.5 Design Guidelines

JSON-LD and Hydra are two building blocks that allow developers to build hypermedia-driven Web APIs. To ensure that APIs created on top of them share REST's benefits in terms of loose coupling, maintainability, evolvability, and scalability, a number of design guidelines have to be followed.

The most important aspect developers have to keep in mind is to not expose implementation details. That means that a change in the implementation on the server should not result in changes in the API it exposes over the Web. In practice that means that developers should introduce an abstraction layer decoupling the internals from the data exposed in the Web API. There exist a number of well-known design patterns to achieve, e.g., the Adaptor or the Composite pattern [90]. The fact that there exists a generic client (that we will present later in this chapter) from the very beginning allows API "usability tests" to be run similar to the usability tests that are typically done for Web sites. This helps to ensure that the API is usable without knowledge of server internals.

From a Linked Data perspective, a vital principle is to reuse existing vocabularies as much as possible. This allows code reuse on the client-side and simplifies data integration. Nevertheless, developers often want to keep full control over the vocabularies they use to provide a unified experience. In such cases, specific concepts should either be sub-typed or declared as being equivalent to concepts in existing vocabularies. This allows more elaborated clients to interpret the data even if they only support the already existing vocabulary. There are significant research efforts to support users in this mapping process which is typically referred to as ontology alignment.

Related to the reuse of already existing vocabularies is the reuse of existing instance data. Just as Web sites typically link to other related Web sites, data exposed by a Web API should link to other related data on the Web; otherwise, services will continue to remain islands in the vast information sea of the Web. This is also a cost effective opportunity for developers to provide their customers with additional data outside of their main business focus. As paradox it might sound, but the more data there is, and the more interconnected it is, the easier it becomes to integrate it with other data.

The data model beneath the exchanged data is a graph. It is therefore possible to serialize the same data in a number of different shapes. When building clients, developers should thus ensure that they do not depend on a specific serialization structure but just on the raw data itself. This is different to how JSON is typically used where the structure is an essential aspect of the contract between the client and the server. Clients have to rely on the structure as that is how the only locally valid semantics are communicated. On the other hand, JSON-LD, with its universal identifiers and unambiguous properties, is much more flexible compared to JSON. Consequently clients are able to use the data independent of the specific serialization structure chosen by the server.

Another important principle to follow when developing clients is to be prepared that everything might change or even break. The machine-readable contract published by the API should be retrieved and analyzed at runtime and not be embedded directly into the client. All the documentations about things such as available operations or possible errors should be seen as hints rather than static contracts. At the moment they are used they might already be outdated and the server might respond in a totally different way than expected. Clients should be able to detect and possibly recover from such errors. As a last resort, the client might need to ask its user for assistance, report an error, or file a bug report.

7.6 Adding Hydra Support to Web Frameworks

Armed with JSON-LD, Hydra, and a small set of design guidelines, we developed a prototype to demonstrate the feasibility of the principles and technologies presented in this chapter. It shows how easily the proposed building blocks can be integrated in real-world systems. The prototype consists of a server exposing an API and a generic client accessing it. The two components communicate by exchanging JSON-LD over HTTP(S).

The server component is based on Symfony2, a Web development framework implemented in PHP. It is, as most other current frameworks, based on a Model-View-Controller (MVC) architecture. MVC is a design pattern that separates the presentation of information from its processing to allow code reusability and separation of concerns. The models represent the relevant entities in the system, the views (in Web frameworks usually templates) are used to create representations of those entities, and the controller is responsible for processing inputs, manipulating the models, and finally returning an updated representation by using the according views. Web frameworks often further modularize the code by dividing controllers into Front Controllers, which handle all requests for a Web site, and Page Controllers or Commands, which are only responsible for certain requests [84]. Therefore, the front controller's job is typically to parse the received HTTP request, extract the request IRI and method, and then pass the control to a specific page controller or command which then, in turn, invokes specific models and views.

While for human-facing Web sites the view layer is crucial and the templates vary widely, it is rarely required in Web APIs. Their view layer is typically much simpler and consists of just a serializer turning the models, i.e., the entities, directly into

representations in a specific format. In object-oriented programming languages entities are usually represented by objects. The prototype we implemented thus extends Symfony2 by a custom bundle, i.e., a plugin, which serializes entities into valid JSON-LD representations. Furthermore, it generates machine-readable documentation based on Hydra describing the entity types and their properties. This allows a client, e.g., to automatically render forms to create valid requests or to provide additional information about the semantics of a specific property.

The serialization component relies, similar to the work of Quasthoff and Meinel [192], on code annotations to control the serialization of entities and the documentation of their types. While this is more complex than simply serializing all public members of an entity, it provides much more flexibility. Not all members should be exposed (all the time) and sometimes transformations, such as converting a numeric identifier into an IRI, might be necessary. The advantage of using annotations is that the information is kept close to the source code it documents, which makes it much easier to keep the two in sync.

Additional to the server component a generic API console or browser was developed for this proof of concept. It is implemented as a single-page Web application using a number of well-known libraries such as Bootstrap, Backbone, Underscore and a slightly modified JSON-LD processor. The JSON-LD processor had to be modified to include additional information in the parsed responses required by the response renderer for tooltips etc.; otherwise a standard JSON-LD processor could have been used as well. The functionality of the client includes the retrieval of resource representations, their parsing and rendering (which includes displaying the related documentation), as well as the invocation of various HTTP operations on embedded hyperlinks, which, in some cases, implicates the dynamic creation of forms to gather the required data to construct valid requests.

7.6.1 Case Study: Issue Tracker

To demonstrate how the just described framework can be used in practice, we implemented a hypermedia-driven Web API for an issue tracker as a case study. This not only allows us to show how such a system might be used in practice but also to describe the underlying framework in more detail.

The first thing to do when developing an API is to define the domain concepts which have to be implemented. In this case, the application domain consists of issues, comments on those issues, and users. Issues have a title, a description, a state (open/closed), a creation date, and a reference to the user who created it. Comments associated to an issue have a description, a reference to the user who created it, and a creation date. Users, finally, have a name, an e-mail address, and a password. Using the user type as an example, we will show how classes can be augmented with the annotations necessary for their serialization and the generation of a machine-readable vocabulary.

As shown in the following extract of the `User` class definition, the fields to be exposed when an instance is being serialized are annotated with a `@Hydra\Expose()` annotation. The password, e.g., will never be serialized, as it is marked as write-only.

Nevertheless it will be documented in the automatically generated vocabulary and be used when deserializing requests. The class itself has a `@Hydra\Id()` annotation which converts the internal identifier (an integer) to a globally valid identifier in the form of an IRI. This is done by referencing the corresponding route which essentially represents an IRI template—in this case `/users/id`. The class also has a `@Hydra\Operations()` annotation which documents the supported operations on this entity. In this example it references routes to replace (update) and delete users.

```
namespace ML\DemoBundle\Entity;
use ML\HydraBundle\Mapping as Hydra;

/**
 * User
 *
 * @Hydra\Expose()
 * @Hydra\Id(
 *   route = "user_retrieve",
 *   variables = { "id" : "id" }
 * )
 * @Hydra\Operations({"user_replace", "user_delete"})
 */
class User
{
  /**
   * @var integer An internal unique identifier
   */
  private $id;

  /**
   * @var string The user's full name
   * @Hydra\Expose()
   */
  private $name;

  /**
   * @var string The user's email address
   * @Hydra\Expose()
   */
  private $email;

  /**
   * @var string The user's password
   * @Hydra\Expose(writeonly = true)
   */
  private $password;

  /**
   * The issues raised by this user
   *
   * @var ArrayCollection<ML\DemoBundle\Entity\Issue>
   * @Hydra\Expose()
   * @Hydra\Collection("user_raised_issues_retrieve")
   */
  private $raised_issues;

  // ... getters, setters, and other methods
}
```

The `raised_issues` property in the code above returns an array of the issues the user raised. The `@Hydra\Collection()` annotation tells the serializer that it should wrap that array in a Hydra `Collection` that can be accessed via the specified route. The alert reader might wonder why there is no variable mapping as in the class' ID annotation. The reason for this is that the serializer is smart enough to create those mappings itself if the IRI template's variables correspond directly to a property of the class. In this case, there is a direct correspondence to the id property. This approach is commonly known as *convention over configuration* and is used to decrease the amount of code/annotations a developer has to write. The same reasoning applies to the automatic code generation of simple CRUD-controllers. All a developer has to do to generate a controller for the just defined `User` class is to invoke the following command in the shell:

```
php app/console hydra:generate:crud --entity=MLDemoBundle:User
--route-prefix=/users/ --with-write --no-interaction
```

This will create a page controller supporting all CRUD operations which listens to requests on the `/users/` IRI prefix. If a developer omits the parameters, a wizard will ask for the required information step-by-step. The code to retrieve the issues raised by a user cannot generated automatically and has thus to be added manually. This is trivial as the code shows:

```
/**
 * Retrieves the issues raised by a User
 *
 * @Route("/{id}/raised_issues",
 *   name="user_raised_issues_retrieve")
 * @Method("GET")
 * @Hydra\Operation(
 *   status_codes = {
 *   "404" = "If the User entity was not found."
 * })
 * @Hydra\Collection()
 * @return ArrayCollection<ML\DemoBundle\Entity\Issue>
 */
public function getRaisedIssuesAction(User $entity)
{
   return $entity->getRaisedIssues();
}
```

The `@Hydra\Operation()` annotation above shows how to document an operation. In this case, the operation would be exposed as `GetRaisedIssuesOperation` and would contain the additional information about when a response with a status code of `404` is returned and what it means in this context (in this case not that no raised issues exist, but that the user does not exist). In the long term, we could envision that a large number of such operations are "standardized" and thus recognized by generic clients—Hydra's built-in CRUD operations are just the beginning. The methods generated by the CRUD controller code generator are automatically mapped to Hydra's built-in operations. This allows the prototype API console we implemented to, e.g., prefill the form generated for a `ReplaceResourceOperation` with the data of the entity.

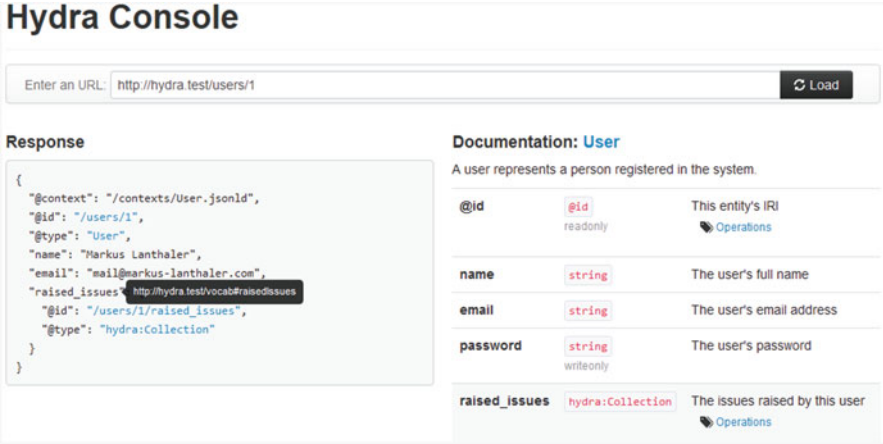


Fig. 7.1 The Hydra console showing a user entity

Implementing the rest of the API is just a matter of implementing the domain concepts and documenting them with the appropriate annotations. The system is then able to automatically generate both a human-readable documentation in the form of an HTML page and a machine-readable vocabulary in JSON-LD for the client. As the response in Fig. 7.1 shows, it also allows the system to automatically serialize the entities returned by the page controllers into JSON-LD documents that look, apart from the link to a context definition and the `@id` and `@type` keywords, almost exactly the same as responses of current JSON-based Web APIs.

The advantage of this approach manifests itself in the fact that it is possible to implement fully generic clients. To demonstrate this, we implemented an API console which allows the user to browse through the API as well as to invoke operations on the various resources. Figure 7.1 shows how a User entity is displayed.

The parsed JSON-LD response is shown in the pane to the left. If the user moves his mouse over a property, a tooltip is shown and the corresponding documentation is loaded in the pane at the right. In Fig. 7.1 the user has his mouse over the `raised_issues` property which expands to the IRI `http://hydra.test/vocab#raisedIssues`, as the tooltip shows. In the background the API console dereferences this IRI, looks for the property definition in the response, displays the documentation of the class associated with the property, and finally highlights the property itself in the displayed documentation. Following links or sending HTTP requests other than `GET` is as easy as clicking on a link and selecting the desired operation as shown in Fig 7.2. If the expected input type for an operation is documented, a form to gather the required data for the creation of a valid request is rendered automatically.

7.7 Conclusions and Outlook

The REST architectural style along with the Linked Data principles builds a foundation to bring many of the key success factors of the human Web to the Web of machines. Instead of building Web APIs with highly specialized interfaces, all the

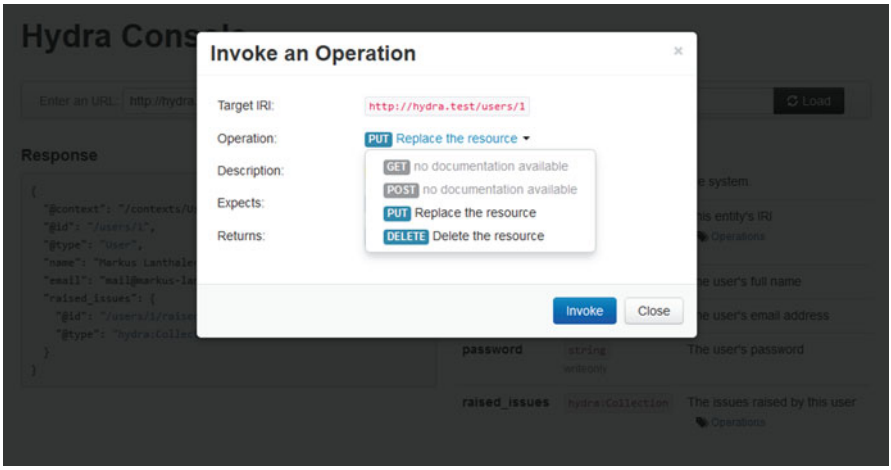


Fig. 7.2 Users can create HTTP requests directly in the API console

modeling happens on a semantic layer completely independent of the underlying serialization format which mitigates the proliferation of custom media types. By using a format such as JSON-LD to serialize the data, a gradual introduction of such a, at first sight, disruptive approach becomes possible. Apart from a few additional properties the responses from such a Web API look almost exactly the same as the ones of current JSON-based APIs. As shown in this chapter, this greatly simplifies the integration into current Web frameworks allowing developers to build on existing infrastructure investments. To the best of our knowledge, no other comparable projects exist that allow the creation of hypermedia-driven APIs in such an integrated manner. The W3C is currently working on a similar approach called Linked Data Platform [216], but so far it does not go beyond standardizing interfaces to implement CRUD services for RDF data serialized in Turtle.

The prototype still misses a few important features due to the early stage of development. Most notably is the lack of support for authentication. To implement that, probably not only the Hydra core vocabulary has to be extended, but also new vocabularies for the different authentication mechanisms have to be created. This modularity in the vocabularies should encourage clients to be modular as well.

In future work we would like to implement a programming library to access such APIs in a programmatic way instead of having to use the user-facing API console shown in this chapter. The long-term goal is to allow a more declarative, goal-oriented access of Web APIs. If the used vocabularies are based on formal semantics (just as RDF's core vocabularies are), it becomes possible to implement reasoners able to infer conclusions which are not expressed explicitly in the data. That, combined with techniques such as hierarchical state machines or behavior trees that allow the creation of reusable blocks of logic, could pave the way for much smarter clients than possible today. By open-sourcing both the client and the server component, we hope to foster the interest of the community to work towards such an ambitious goal.

Chapter 8
RestML: Modeling RESTful Web Services

Robson Vincius Vieira Sanchez, Ricardo Ramos de Oliveira and Renata Pontin de Mattos Fortes

8.1 Service Oriented Computing

In the last decade, it was possible to witness a significant change on IT applications, which evolved from distributed computation paradigm to a service-oriented computing (SOC) paradigm, which is gaining prominence as an effective approach to integrate applications in distributed heterogeneous environments [154] in order to solve the problem of lack of communication between different information systems present in corporate environments. Because of this interest, studies related to Web Services have increased.

RESTful web services have emerged as a way to simplify the development of web services, following the principles of REST architecture, to ensure the same portability achieved by the Web and making it easier to publish and utilize the web services [184].

The already existing patterns, which are widely used on the Web, such as HTTP, XML, URI, and others, were used for the development of RESTful services, avoiding excessive standardization present in SOAP Web services.

Among the advantages present in REST architecture, the most frequent are presented next ones are [184, 196]:

- **Simplicity in development:** by using Web patterns, and preventing over-standardization;
- **Scalability:** by allowing support to cache, clustering and load balancing;

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

R. V. V. Sanchez (✉) · R. R. de Oliveira · R. P. de M. Fortes
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo,
Avenida Trabalhador São-carlense, 400 – Centro CEP, São Carlos, SP 13566-590, Brazil
e-mail: robsonvinicius@gmail.com

R. R. de Oliveira
e-mail: ricardoramos@icmc.usp.br

R. P. de M. Fortes
e-mail: renata@icmc.usp.br

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_8,

- **Possibility of multiple representations of a resource:** it allows the use of light formats to improve the performance;
- **Stateless communication:** it leads to independent requests, allowing bookmark of URIs and improvements of scalability;
- **Hypermedia as the engine of application state (HATEOAS):** it allows to connect the resources by using hyperlinks;
- **Uniform Interface:** it uses the methods of HTTP protocol to perform the operations on the resources in a uniform way.

While efforts are being devoted to simplify integration of information systems using Web technologies, new approaches to modeling Web applications have been proposed to systematize the development of these applications, as well as to make the development platform-independent [203]. In order to make it possible, the new approaches use, in general, the model-driven development (MDD) paradigm.

8.2 Model Driven Development

Model Driven Development (MDD) is a method that focuses on the construction of models and on the definition of transformations which automatically generate source code (or other models) from artifacts [229], allowing engineers and developers to work on a higher level of abstraction during the software development. Using a model-driven approach it is possible to simplify and to systematize the various activities that constitute the software development lifecycle [105].

The MDD approach assumes that models are essential artifacts in the software development process and it is based on the models that applications will be built, while in a conventional software development, modeling is used mainly to facilitate analysis of the problem, thus the models are used as a support to developers in the later stages of a project. Thereby, in a conventional software development, the developers implement the code manually, which often makes the models previously created inconsistent with the actual software developed. On the other hand, by using MDD, because the models are responsible for the code generation, they are always updated facilitating the maintenance and documentation of software.

The construction of models as an abstraction of the software being implemented allows the developers to focus on their efforts on modeling the problem domain and not on a programming level solution [203], hiding the possible implementation details, which increases the portability and reuse properties of the software.

The advantages of MDD are achieved due to the possibility of automatically generating source code, by transforming the models and artifacts in general, and preventing the developers of performing repetitive tasks for writing source codes. Among the advantages pointed out by [122, 146], we mention:

- **Productivity:** increased productivity in the process of software development. Mainly due to reduction of repetitive tasks by the developers and also to the possibility of reuse;

- **Correctness:** transformations used by MDD for generation of source code prevent to perform repetitive and manual tasks by the developers, avoiding accidental errors such as typo errors, thus ensuring a source code more accurate and error-free documents;
- **Portability:** different transformations are applied to a model capable of generating code for different platforms, thus favoring portability;
- **Maintainability:** in conventional development process, the maintenance or evolution of software is performed directly in the source code, which can spend a great effort that is almost comparable to the effort spent during development. On the other hand, in MDD the maintenance is performed on the models and code is then re-generated from them;
- **Documentation:** in MDD the models are the main artifacts of the software development process and for that reason the artifacts do not become out of date since the code is generated from them (the models). Thus, the system documentation is always kept updated;
- **Communication:** the models are easier to understand than the source code, which makes communication easier between the developers and the other team members;
- **Reuse:** in MDD, reuse of models means that the code can be re-generated automatically for a different context, without the developers having to perform manual tasks, which does not occur in conventional development.

A domain specific language (DSL) is a machine processable language whose terms are derived from a model of an specific domain, i.e. they are used for special field and capture precisely the semantics of this field [230]. In a very simple definition, DSL can be considered a small, usually declarative language, focused on a particular problem domain [235]. Therefore, these languages are created specifically to solve problems in this area.

At the level of the modeling languages, UML (Unified Modeling Language) is one of the most popular languages, widely used for modeling systems based on the object-oriented paradigm. UML is a modeling language considered of general purpose, but can be specialized by extension mechanisms, known as Profiles, making it a domain-specific language. As an example, we have the UML *Profile For Enterprise Application Integration* [175], specialized for applications integration, or even the UML *Profile for Modeling and Analysis of Real-Time and Embedded Systems* [177] that is specific to the development of embedded and real-time systems.

This chapter aims to present a DSL capable of modeling RESTful Web Services and generate source code by means of transformations according to the principles of the model-driven development.

8.3 Related Work

Web Engineering focuses on the methodologies, techniques and tools that are the basis for the development of web applications, and therefore it is considered a domain in which the MDD approach can be useful, especially in the treatment of problems related to evolution and adaptation of web systems [126].

Among the existing approaches (e.g., [205, 222]), three of them will be analyzed: UWE [127], WebML [48] and OOWS [234].

UML-based Web Engineering (UWE) is an approach proposed to help the modeling of web applications. Its main goal is to reinforce the separation of concerns. There are different models for presentation, contents, structure hypertext and processes [133]. The approach has a metamodel based on UML. Thus, a major advantage of the UWE metamodel is related to the fact that it remains compatible to the MOF (Meta-Object Facility) meta-metamodel, used for the definition of UML, which was defined by the OMG and therefore follows the approach of Model Driven Architecture (MDA [126].

Besides the previously mentioned advantage of UWE approach, another important aspect is the possible integration with modeling tools that work with UML Profiles, becoming more familiar to the developers that are familiar to model using UML. However, despite these advantages the UWE metamodel has been developed for modeling Web applications and has no specific elements for modeling Restful Web Services.

Web Modeling Language (WebML) was defined in 1998 as a conceptual model for specification of web applications with large capacity data [47], i.e. its primary goal was to generate or manipulate contents stored in one or more data sources [152]. While other conceptual models focus more on the early stages of the development process, e.g. requirements gathering, WebML focuses in the later stages, such as the project and the implementation's phases.

The WebML is extensible and allows the definition of custom modules, which can be used to include new operations in the web application, for instance RESTful Web services. However, RESTful services provided by WebML do not conform to the strict REST architecture specification, producing a hybrid REST-RPC architecture (RPC — *Remote Procedure Call*). Additionally the WebML is not defined in an specific meta-model, which makes impossible the use of custom transformation languages, such as the ones that are used in MDA.

[234] proposes a Model-Driven approach to integrate REST APIs in Web applications using Web engineering methods. The authors defined a meta-model that describes RESTful Web services, however the approach has the goal of integrating Web applications with REST APIs, and not elaboration of RESTful Web services.

In this context, we present the domain-specific language RESTML, which has the objective of modeling RESTful Web services according to the REST architecture strict specification. RESTML is defined in an UML based meta-model which can produce domain models that can be transformed into other models and source code, using the MDA approach proposed by OMG.

8.4 RestML

In the context of MDD, this chapter aims at describing the definition of a DSL able to represent the static and behavioral characteristics of RESTful web services in order to create models that could help the developers to specify the corresponding representative web services. The DSL must therefore be complete enough so that

the generated models based on the DSL may be used as input for the automated transformations that generate the source code.

The DSL named RestML is defined as a UML Profile since its main motivation is to use the Model Driven Architecture approach [174] for model-driven development, which would permit to build models as an extension of UML 2 and would transform them using languages for model transformation as QVT (Query/View/Transformation).

8.4.1 UML Profile

UML provides a set of extension mechanisms (stereotype, tagged values and constraints) in order to specify its elements, which allows the elaboration of custom UML extensions for specific domains (PIMs) or platforms (PSMs) [88]. These extensions are grouped in an UML Profile which allows the customization of any meta-model defined in MOF.

The extension mechanisms of the UML Profile are available by means of an UML package, which contains the stereotype ≪profile≫. Two UML Profiles were defined in RESTML: one for defining Platform Independent Models for representing RESTful Web Services and another one for defining Java Enterprise Edition (JavaEE) models (PSM) [178].

8.4.2 Modeling RESTful Web Services

The RESTML language was implemented as an UML Profile for modeling RESTful Web Services, in order to include information in UML models according to the proposal established in the REST architecture style. Stereotypes and tagged values were elaborated to map the architecture constraints defined in REST and allow that the models, which are created using this Profile, be automatically transformed into other models.

Thus, RESTML has the goal of providing developers with a modeling tool for RESTful Web Services, which conform to the strict definition of REST architectures and is free of platform specific details. These models can be then specialized for different platforms.

Figure 8.1 illustrates how the diagrams can be built to express the characteristics and relationships among different components in a RESTful Web Service. The defined components are:

- Resource: receives external requisitions by means a uniform interface;
- Service: component which is responsible for the business logic in the application;
- Representation: data format for one specific resource;
- Exception: identifies problems that might have occurred during the execution of a service;
- Data access: components that are capable of accessing data that is external to the application.

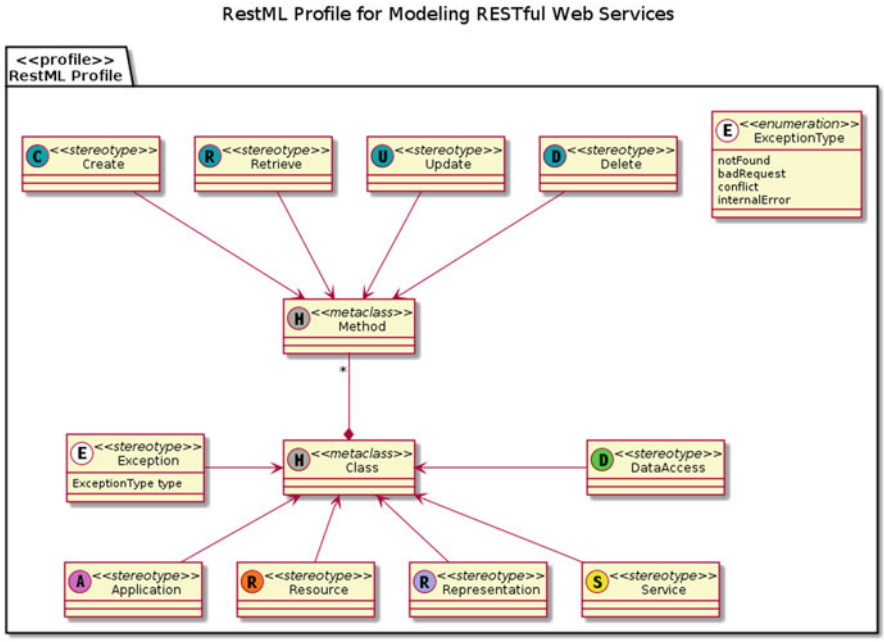


Fig. 8.1 RestML Profile for modeling RESTful Web Services

Each of these components has a distinct objective in the REST architecture proposed in the RESTML language and need to be correctly represented so that the transformations can be performed in all components. The different types of representation are modeled using the stereotypes defined in the UML Profile for each of the components of an application.

The stereotypes introduce a new model element as an extension/classification of a previously defined element [7]. In this context, the stereotypes should be applied in class and sequence diagrams. As the stereotypes are applied to a model element, its property values are attached to the element and can be retrieved as tagged values, which are stored in the element in a key-value pair.

In the RESTML UML Profile ten stereotypes were defined:

1. Application: identifies a resource that represents the application. This stereotype should be applied only in the classes definition;
2. Resource: identifies a resource of the system. This stereotype can be applied in the classes definition;
3. Create: identifies the creation of a new resource. This stereotype should be applied only for methods;
4. Retrieve: identifies a search resource. This stereotype should be applied only for methods;
5. Update: identifies a resource update. This stereotype should be applied only for methods;

6. Delete: identifies the exclusion of an existent resource. This stereotype should be applied only for methods;
7. Service: identifies a component that represent the business logic, therefore it should describe a transactional behavior;
8. Representation: identifies a data format that represents a resource of the application;
9. Exception: identifies a component that is responsible of informing an error condition on the system;
10. DataAccess: identifies a component that is responsible of accessing external data on the application, for instance, database, files, RMI, Web Services, among others.

This UML Profile definition provides developers with a platform independent modeling tool for RESTful Web Services. Next section describes a platform specific UML Profile definition for JavaEE.

8.4.3 RESTful Web Services in JavaEE Platform

The *RestML Profile for Modeling RESTful Web Services* aims to model the common characteristics of RESTful web services independently of platform, however it is important that the models created with this profile are transformed into specific models of a platform so that the models can be used to automatically generate source code.

To fulfill this approach, this project proposes another UML Profile called *RestML JavaEE Profile for Modeling RESTful Web Services*, which aims at providing Java Enterprise Edition specific platform models (PSM) that can be generated from *RestML Profile for Modeling RESTful Web Services* models (PIM).

In Fig. 8.2 we illustrate the definition of *RestML JavaEE Profile for Modeling RESTful Web Services*.

Figure 8.2 illustrates that there is a great number of stereotypes and tagged values that are required to specify details of the platform in which the application will be implemented.

The transformation between the two profiles described in this chapter will be discussed in more detail in Sect. 8.4.8.

8.4.4 MDA Approach

Using the MDA approach, three types of models can be constructed:

- CIM (Computer Independent Model): also called domain model, it does not show details of the structures of the system, only a vocabulary familiar to professionals in the field, which is used to perform the specifications. Thus, the professionals do not know the models and artifacts used to create the functionality of the system,

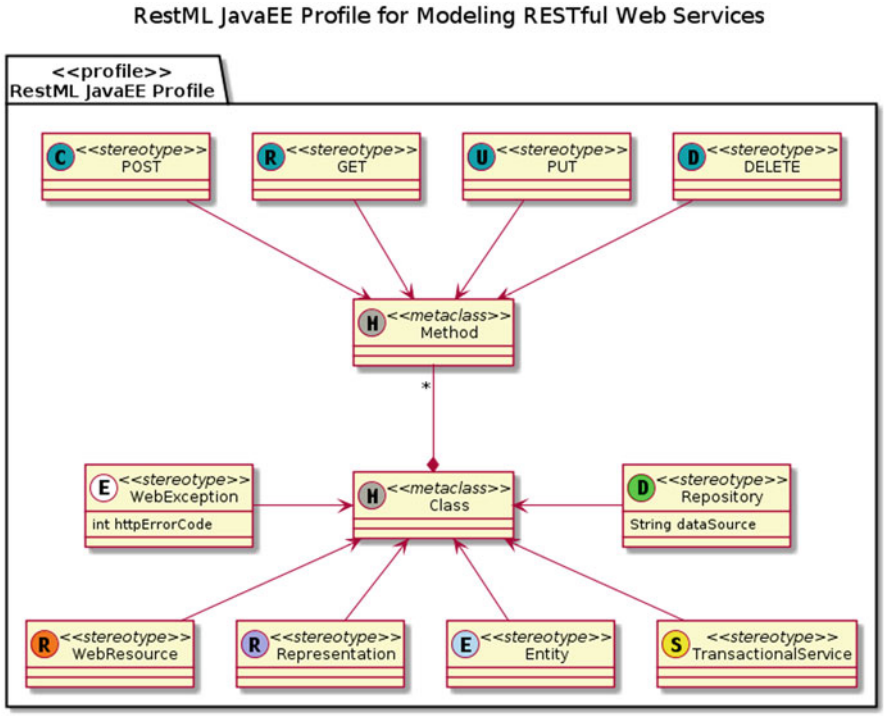


Fig. 8.2 RestML JavaEE profile for modeling RESTful Web Services

knowing only their requirements expressed by the CIM. Hence, this model is important to realize the connection between the experts' work in the field (domain experts) with the experts in design, responsible for the construction of artifacts.

- PIM (Platform Independent Model): consists of a vision of the system that is platform independent, focused on the functionality of the system, hiding the details of each specific platform.
- PSM (Platform Specific Model): a PSM combines the specifications of the PIM with the details of an specific platform.

Hereafter, the requirements of a scheduling system for boards at an university domain will be modeled as services of a web application as a proof of concept, using RestML. The transformations that are applied to the models in order to generate Java Programming language source code automatically are also defined.

8.4.5 Case Study: AgendaWS

An important activity in the final year of undergraduate courses consists of the presentation of works that students individually develop as a concluding research. And at the end of the period, they usually have to present the results of this research to a board constituted by lecturers. The need for communication among the students

and the lecturers in order to collect information to schedule convenient and available dates for presenting their works was the main motivation for the development of a group calendar system (GCS).

GCS are calendars that can be shared in the web and aim to support the need of schedule information integration for personal use or for use by a particular group. In the context of the undergraduate boards for presentation of their final researches, it was proposed an evolution of a already existing GCS, with the inclusion of new features and allowing the data to be accessed by different applications present in the university environment.

To address these requirements, a web application containing "AgendaWS" RESTful web services was proposed to be developed. The web services needed to be capable of providing a simple and uniform interface allowing integration with other applications. The "AgendaWS" RESTful web services were then developed using the paradigm of Model-Driven Development through the MDA approach with the use of language RestML. To support the development of the models, the RestMDD tool was used, which besides enabling the design of models according to the RestML language, it also performs the necessary transformations between them.

Figure 8.3 shows the life cycle of the MDA. Initially in "analysis" phase, the domain specialist receives the requirements for developing the application and generates a CIM from the requirements. After this step, in "design" phase, the specialist in design turns the CIM to PIM models needed to represent the functionality of the system as proposed earlier. Then, PIM can be transformed into one or more PSMs that possess the information specific to each platform. Next, "coding" phase, the PSMs are submitted to a transformation to generate the source code of the application.

In the next sections each of the models developed for designing the application AgendaWS will be described, and how these models represent the relevant characteristics of RESTful web services.

8.4.6 The Domain Models (CIM)

The first step in developing an application using the MDA approach is to elaborate the domain models whose main objective is to represent the application requirements through a common vocabulary to specialists in domain and to experts in design. In that way, modeling the behavior of the system is possible without the need to know the artifacts required to implement the functionality. In RestML, three types of domain models were defined:

1. Use Cases
2. Use Cases Diagrams
3. Activities Diagrams

The models are usually defined by experts in the field based on a requirements document of the system and serve as input for the construction of the platform independent models.

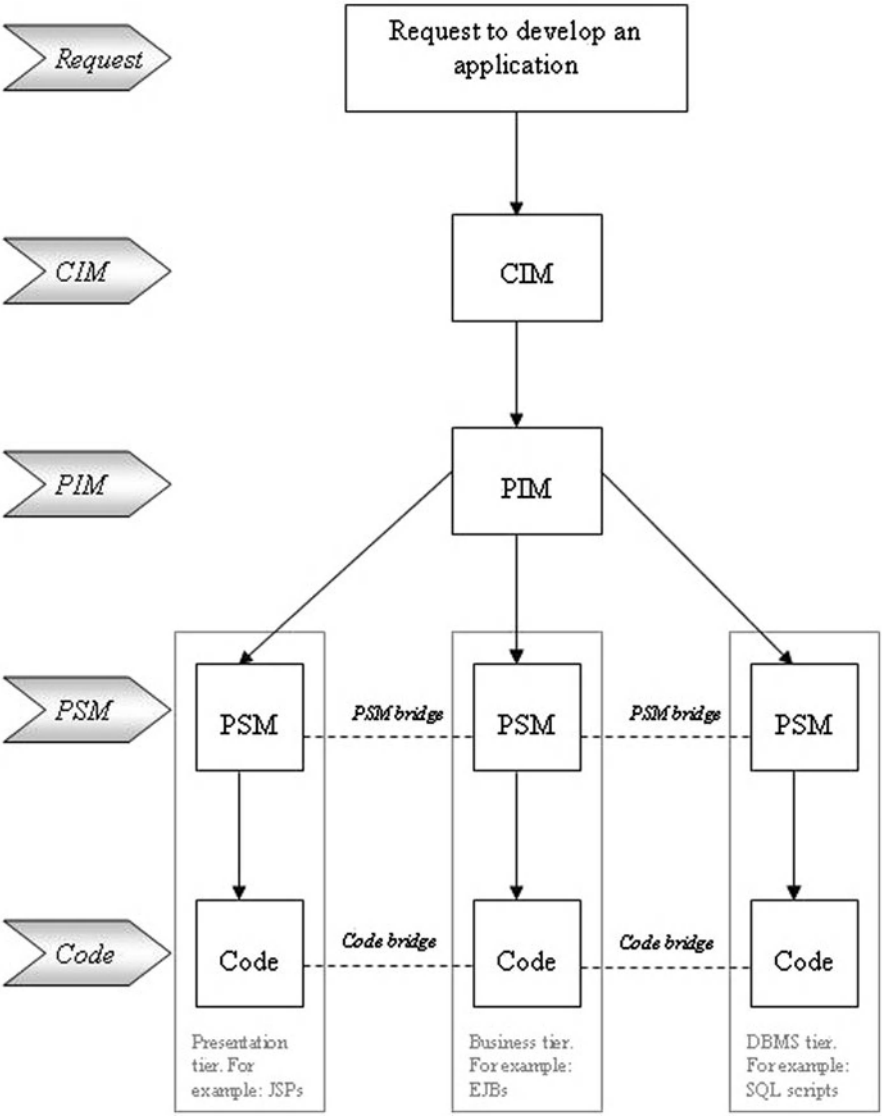


Fig. 8.3 The MDA Life Cycle [228]

8.4.6.1 Use Cases

An Use Case is a rather description of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal [53]. In a simple definition, a use case describes how an user interacts with the system and how the system answers to him [198].

By means of modeling using use cases, we identified the events that must be realized so that a requirement is addressed and the interactions that the actors need to perform in the system. Thus, use cases can be considered the foundation of the dynamic behavior of the system.

In RestML the development of RESTful web services depend on the modeling of a use case for each task that can be performed by users. Adopting the MDA approach, the use cases are first-class artifacts and through them, the other models can be built.

In this context, the contents of a use case model should contain the following components:

- **Name:** name of the use case, utilized to identify it.
- **Description:** a brief summary of the purpose of the use case. In the description references to the requirement to which the use case serve can be found.
- **Actors:** relation of actors that can perform this use case.
- **Type:** identifies the type of use case. The allowed values are: Create, Retrieve, Update and Delete. This type will be important to identify which method of the uniform interface of RESTful services should be used.
- **Resource:** resource name that will be the gateway to the execution of the use case.
- **Pre-condition:** initial state of the system that is required so that the use case can be executed.
- **Main Flow:** contains the tasks that the actor must perform as well as the answers produced by the system.
- **Alternative Flow:** takes into account the exceptions of the main ow.
- **Post-condition:** expected result after the execution of the use case.

In Fig. 8.4 there is an example of an use case of the system AgendaWS that includes the registration of a university.

The REST architectural style enables all communication between client and server to be performed using a uniform interface. Therefore, the use case models should be inform the use case type and its resource, so that it becomes possible to relate the use case with the uniform interface of the resource.

Once the use cases have been modeled, we create a use case diagram, obtaining a simple, visual representation of existing use cases and also their relationships with actors.

8.4.6.2 Use Case Diagrams

An use case diagram is able to show the relationships among actors and use cases within a system [10]. In general is used for:

- Providing an overview of the requirements of a system;
- Communicating the scope of a project;
- Modeling the analysis on the requirements in the form of a systemic use case model.

Name: Create University

Description: This use case aims to create a new university in the system.

Actors:

- Administrator

Type: Create

Resource: University

Pre-condition: The user must be logged in.

Main flow:

1. The administrator provides the name and acronym of the university;
2. System validates that the data provided are correct. If have any invalid information, go to the alternative flow 1;
3. System validates that already exists another university registered with the same acronym. If there is, go to the alternative flow 2;
4. System records the information;
5. The administrator receives a successful response with the data that were registered.

Alternative Flows:

- **Alternative Flow 1**

  The administrator receives an error response indicating which fields are invalid or incomplete.

- **Alternative Flow 2**

  The administrator receives an error response indicating that already exists another university registered with the acronym informed.

Post-condition: The university must have been recorded in the database.

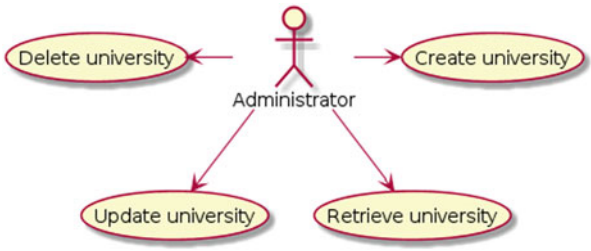
Fig. 8.4 Use case: Enrollment of a university

In the scope of the RestML language, the use case diagram will be constructed from the use case definitions and their respective actors, so by automated tool "RestMDD".

It is possible to create different use case diagrams representing independent parts of the system. Then an example of a use case diagram of the generic operations related to maintenance of AgendaWS implementation is shown in Fig. 8.5.

The use case diagram is not used directly to the creation of other artifacts and models of an application, but it has a fundamental role in the documentation of the

Fig. 8.5 Use Case Diagram



system and also for the communication between team members. We argue that the abstraction of models must always be liable to reading and understanding by the application's developers.

8.4.6.3 Activity Diagram

Activity diagrams are usually applied for modeling business processes, for capturing the logical description of a single use case or to model in details the logic of a business rule [11].

The activity diagrams can be classified as domain models since they do not offer any information about how the activity will be implemented, hiding the technical details that are irrelevant to the solution design. The activity diagrams only identify

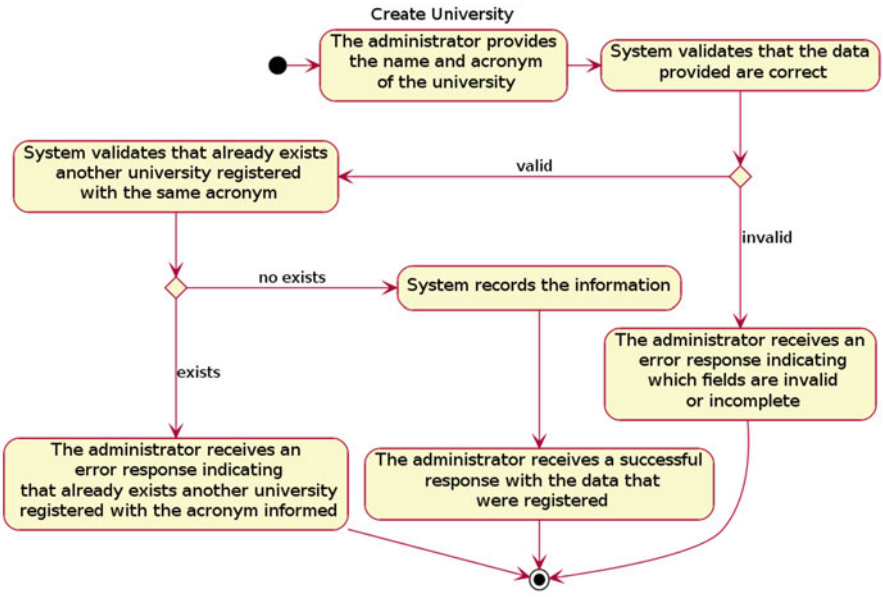


Fig. 8.6 Activity Diagram

which activities should be performed by the use case to happen, without informing the inner workings of each of these activities.

In the scenario described in this chapter, an activity diagram will be constructed for each use case previously registered. The activity diagrams should be generated automatically by transformations applied to use case models, in such a way that each basic event flow and also the alternative flows are transformed into activities at the diagram.

A representation of the structure of the diagrams is defined in XMI format (XML Metadata Interchange) [176], a format for representing models in XML can do the exchange of UML and other models based on MOF. Thus, XMI defines a mapping of UML / MOF to XML, which is capable of integrating tools, applications and repositories.

In Fig. 8.6, an activity diagram generated by the tool RestMDD from the use case model defined in Fig. 8.4 is illustrated.

Therefore, we can model the dynamic behavior of an application in its different usage scenarios and by means of the behavior, designers can model the platform independent models, which will document the internal behavior of each software component. At the moment the development of the software moves from Analyze phase to Design phase.

8.4.7 Platform Independent Models (PIM)

In short, a PIM is a formal view of the functionality and structure of a software system without reference to any particular computing platform [157]. These types of models are created from transformations applied to domain model and they model how the implementation of system functionality will be, but without containing specific aspects of a platform or programming language. Accordingly, we can obtain different models of the software solution design portable to different environments.

In the context of language RestML two types of platform independent models were defined:

1. Class Diagrams
2. Sequence Diagrams

Each one of these models have different goals in modeling, the class diagrams being responsible for the definition of the application given its static structures such as resources, entities, representations, among others. And the sequence diagrams define the dynamic behavior of the system showing the message exchange between the various components of the architecture.

8.4.7.1 Class Diagram

Class diagrams are used for a variety of purposes, from conceptual modeling until detailed design modeling. The diagrams show the system's classes, their inter-relations (including aggregation, inheritance and association) and the operations and attributes for each class [9].

In the registration of the use cases the analyst tells the resource to which the use case is associated, as well as its type. Based on this information, it is possible to generate a class diagram defining the classes for resources and their methods. It should is applied the stereotype ≪Resource≫ for these classes of resources and one of the following stereotypes: ≪Create≫, ≪Retrieve≫, ≪Update≫ or ≪Delete≫ are applied to the methods. For each resource defined in the use cases, a resource class must be created. Furthermore, every use case has a type that will served to include a method in the resource class.For the use cases of type Registration, a method with ≪Create≫ stereotype must be created; for the use cases of type Change a method must have the stereotype ≪Update≫; for the use cases of type Exclusion the stereotype ≪Delete≫ should be used and finally the use cases of type Query the stereotype ≪Retrieve≫ should be used.

The resource class will become responsible for receiving the request from customers, validate the input parameters, invoke a method of the class of service that will take care of all the business logic and finally build the response to be sent to the user. Therefore, for any resource class, a class of service must also be created. This service class will be responsible for the business logic of the process and must contain the stereotype ≪Service≫. The name must match the methods in the two classes.

Each resource requires that a class is created and this class will define the representation containing all the attributes representing that resource. The representation class must have the stereotype ≪Representation≫. In order to access the external data the application needs to define a class of data access with the stereotype ≪Data Access≫.

Finally, in the use cases execution faults can occur and should be reported with different codes and error messages. Therefore, classes that represent these failures need to be created (≪Exception≫).

In Fig. 8.7 the class diagram created from the use cases defined in Fig. 8.4 is shown.

8.4.7.2 Sequence Diagram

The sequence diagrams model the logic flow within the system in a visual manner, allowing the documentation and validation of its logic. The sequence diagrams are very popular artifacts for modeling dynamic with the focus on identifying the system behavior [11].

In synthesis, sequence diagram is one of the UML tools used to represent interactions between objects of a scenario, performed by the methods, emphasizing the sorting time at which messages are exchanged between objects.

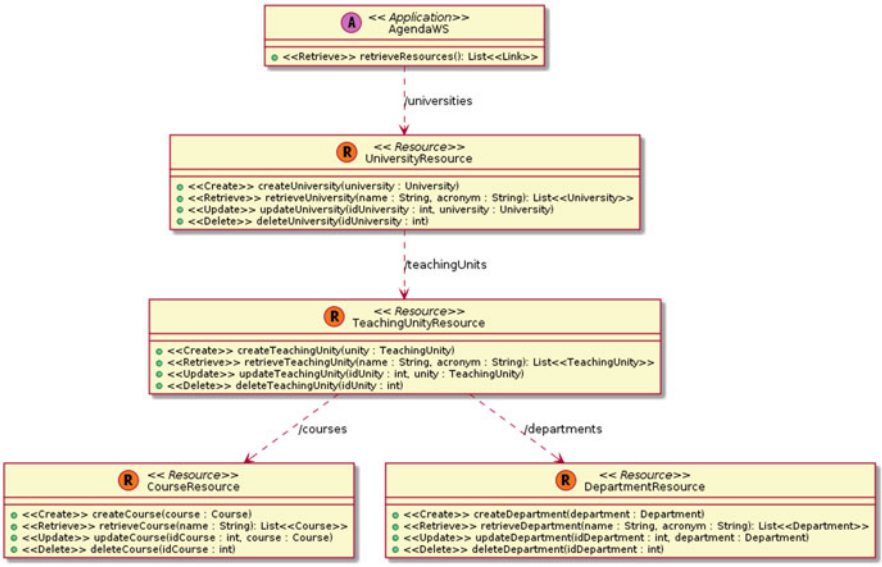


Fig. 8.7 Class diagram

The definition of a sequence diagram using the language RestML will be held in a semi-automated manner with information coming from the activity diagram created in the Analysis phase.

For each activity of the activity diagram, it is necessary the definition of what components will perform this activity and which messages will be exchanged between them. The developer, using only the classes identified as Resource, Service, Data Access and Exception, must set this message exchange. This modeling has some restrictions:

- It is allowed only one component of each type;
- The actor must start the flow of messages within the diagram and should only communicate with the resource;
- The resource can only send messages to your service, to the actor or himself (self message);
- The service communicates only with the resource and data access class;
- The data access class communicates only with the service.

After completing the modeling, the environment RestMDD will generate a sequence diagram as shown in Fig. 8.8.

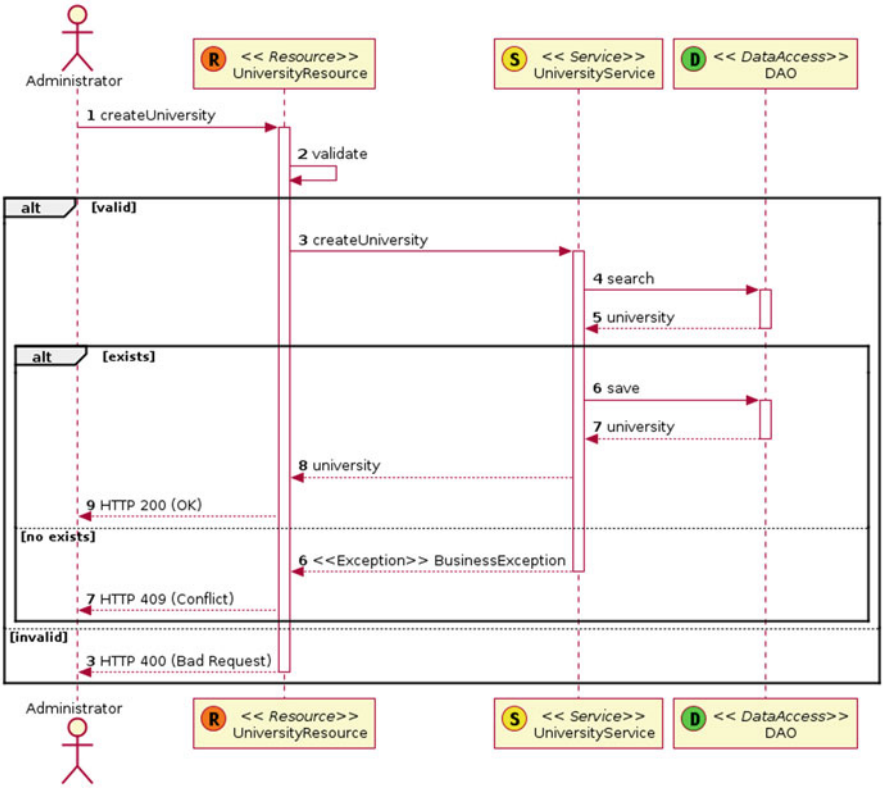


Fig. 8.8 Sequence diagram

8.4.8 Platform Specific Model (PSM)

The next step in the process of software development using the RestML language is to transform the platform-independent model (PIM) into a platform specific model (PSM). A PSM combines the specifications obtained in PIM with the details that are specific to a platform, i.e., the model represents a view of the system placed in the context of a specific platform.

Thus, the language RestML proposed only one transformation to the Java Enterprise Edition platform which includes several API's for Web development together with the Spring Framework, which provides a container capable of performing inversion of control (IoC) and dependency injection, and JAX-RS that provides API's for development RESTful Web Services.

This transformation basically consists in applying RestML JavaEE Profile for Modeling RESTful Web services in existing classes. However, it is necessary that some activities are conducted:

- Classes that contain stereotypes ≪Application≫ and ≪Resource≫ should receive the stereotype ≪WebResource≫, identifying these classes as application resources;
- Classes that contain the stereotype ≪Representation≫ will have two stereotypes ≪Representation≫ and ≪Entity≫. The first identifies that this class is a representation of the resource and the second stereotype indicates that this class is an entity of the entity-relationship model of the database accessed by the application;
- Classes that contain stereotype ≪Service≫ need to receive the stereotype ≪TransactionalService≫, identifying a transactional business component;
- The exceptions with the stereotype ≪Exception≫ will be ≪WebException≫. The tagged value *httpErrorCode* should contain the HTTP error code according to its type. Exceptions like "*notFound*" must have the code 404, type "*badRequest*" must have the code 400; already type "*conflict*" must have the code 409 and lastly exceptions of type "*internalError*" with code 500;
- The data access class with the stereotype ≪DataAccess≫ needs to receive the stereotype ≪Repository≫. The tagged value dataSource must contain the name of the data source configured on the server;
- The methods with stereotypes ≪Create≫, ≪Retrieve≫, ≪Update≫ and ≪De-lete≫ will become ≪POST≫, ≪GET≫, ≪PUT≫ and ≪DELETE≫, respectively.

In the diagram there are no significant changes in the exchange of messages between application components. Classes transformed from PIM to PSM continue exchanging messages in the same temporal order defined in PIM.

8.4.9 Source-Code

After all domain specific models are built, the next phase of software development is coding, which is performed by transforming these models into source code. The transformation is performed with the support of open-source framework AndroMDA.

In short, AndroMDA can get different models (UML models usually produced by CASE tools and stored in XMI format) combined with various plugins (cartridge and translation-libraries) and produces different types of components [15]. Is possible to create components in several programming languages and platforms just writing or customizing plugins that support the transformation of XMI models into source code.

Plugins in the AndroMDA framework were built, for the RestML language, which made possible to transform the diagrams, created during design phase, into source code in Java language for the platform using JavaEE frameworks Spring and JAX-RS.

8.4.10 Packaging and Deployment

Once all the source code was generated in Java programming language it is important to package these artifacts so that they can be run on a Web server as a library or even within an already existing web application. Thus, there are two possibilities:

1. A WAR file: creation of a Web Archive (WAR) that can be implemented directly within a Servlet container;
2. A JAR file: creation of a Java Archive (JAR) that should be used in other applications as a library Web

The packaging is accomplished by the tool RestMDD, according to the preference of the developer.

8.5 Final Remarks

In this chapter an approach to Model Driven Development for the construction of RESTful web services through a domain-specific language called RestML was described. The RESTful Web services were defined through a UML profile and designed following the principles of MDA architecture.

It was also documented all models created in each phase of the development of a RESTful web service and all the features that these models must possess. Furthermore, it was shown the relationship between domain models, platform independent models, platform specific models and source code.

Finally, it was shown both forms of packaging and deployment of the generated source code.

Acknowledgments Acknowledgments to USP and Fapesp.

Part II
Practical Applications

Chapter 9
A Lightweight Coordination Approach for Resource-Centric Collaborations

Morteza Ghandehari and Eleni Stroulia

9.1 Introduction

Today we are witnessing an abundance of technologies, conceived to support the development of web-based applications in general, and web-based service-oriented systems in particular. In the service-oriented paradigm, processes are typically supported through service orchestration; the various steps of the process are delegated to services, and service orchestration is employed for coordinating the operations. Traditionally, the steps are implemented by WSDL/SOAP web-services operations, and the process model is specified using WS-BPEL as the de-facto standard language for web-service orchestration; furthermore, the process itself can be published as a new service, implemented on a WS-BPEL engine.

More recently, the REST (*Representational State Transfer*) [76] architectural style has emerged as an alternative approach for development of web-based systems and there are three reasons for this phenomenon.

1. First, the REST approach is conceptually and syntactically simple; it relies on the HTTP protocol, with XML (or JSON) as the exchange format for the payload data, and a simple syntactic style for formulating HTTP requests as traversals of the XML schemas of the underlying resources.
2. This simplicity makes the related learning curve smoother, since developers can easily migrate from standard web-based development to exposing their systems as resources accessed through REST APIs.
3. The increased availability of interesting information and simple development tools (some based on the demonstrational-programming paradigm) through REST APIs has spurred the interest of web users to develop information mash-ups.

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

M. Ghandehari (✉) · E. Stroulia
Department of Computing Science, University of Alberta,
2-21 Athabasca Hall Edmonton, Alberta T6G 2E8, Canada
e-mail: morteza.ghandehari@ualberta.ca

E. Stroulia
e-mail: stroulia@ualberta.ca

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_9,

Given the conceptual simplicity of the REST style, many service providers are now publishing their services as REST services, instead of (or in addition to) WSDL/SOAP web-services. At the same time, governments, following the "open data" movement, are making rich information repositories publicly available. As a result, the number of information resources accessible through REST APIs is increasing and so is the number of web-based interactive tools for using and manipulating these resources. The question then becomes how to coordinate the activities supported by these various tools.

In this work, we focus on resource-centric collaborations. The term *resource-centric collaboration* refers to a type of human-intensive workflows[1] in which a group of people work together on shared information resources. The information is available to the collaborators in a resource-centric environment and can have various types such as XML data, text files, or bibliographic references. The objective of the collaborative activity is to develop and manage the various resources by taking turns editing them, annotating them with metadata, and evaluating their degree of progress and completion. Examples of this type of collaborative work include coauthoring a scholarly publication, producing a project report, and coding/annotating an original text with metadata.

Let us consider a simple project-report coauthoring collaboration where a group of students work together to produce a report on their team project. A team member may develop the first version of the report and provide it as a resource on the Web. Then other team members may take turns editing the report, with the turn-taking order being ad-hoc, or possibly depending on the members' roles in the team. Each team member, when she is done with her turn, indicates the availability of the resources (the report and its sections) to the team, possibly through an explicit notification mechanism. At some point, a team member, who has been designated to regularly inspect and assess the quality of the report, may decide that the report is no longer a "draft" but has been "completed" and submit the report to their supervisor for approval. The supervisor receives the report and evaluates it. If the supervisor approves it, the resource is "published"; otherwise, the report is sent back to the team, possibly with specific comments to be addressed, for further editing.

Resource-centric collaborations, of which the above project-report coauthoring story is an instance, share several common properties.

- **They involve many "people" activities**: While various editors and tools are involved in the overall activity, the collaborative activity is initiated and driven mostly by the people participating. In the example discussed above, it is the responsibility of each team member (a) to receive the URL of the resource and start editing, when they are notified by the system, and (b) to decide when to release the resource to the rest of the team. Contrast this with automated workflows where the various steps are performed by automatically invoked software services that explicitly signal their completion with their return parameters.

[1] In this paper, we use the terms "workflow" and "process" interchangeably.

- **Their steps are loosely ordered**: Resource-centric collaborations are semi-structured. In other words, the control over the various tasks is non-deterministic, and in many cases, a particular task can be performed any time. In our example, although the report may consist of specific sections, the order in which these sections are edited is unlikely to be predetermined; any section can be edited any time and it is only important that all the sections are written before the report can be considered "completed". In contrast, in automated workflows, although there may be alternative control-flow paths, each one is annotated by an explicit condition on when it is taken.
- **They have simple structure**: The process models of resource-centric collaborations usually do not have complex control elements, computation, or data transformations. They usually involve the evaluation of conditions (e.g. to assess whether a particular person may access the resource), simple service calls (e.g. to receive notifications of activities that have occurred through the users' interactive tools, and to generate notifications for other users about the state of the process), and value assignment (e.g. to manage the transition of the resource through its lifecycle phases). In contrast, web-service orchestrations must support the mapping of complex parameters through services, the maintenance of global variables, and the evaluation of complex control structures. Therefore, languages such as BPEL [227] are too complex and a simpler language would be more appropriate for specifying them.
- **They have undemanding performance requirements:** Finally, resource-centric collaborations do not usually have complex or strict performance requirements. For example, automated workflows may indicate upper limits in the response time of a service (and upon its expiry the middleware may invoke a fall-back alternative). In contrast, resource-centric collaborations may need to conform to deadlines for the overall completion of the coordinated task, but it is unlikely that any of the individual activities involved is time-critical.

Current solutions in support of resource-centric collaboration (Fig. 9.1) include tools such as collaborative editors, document repositories and software version-control systems. These tools usually provide some basic coordination support, like controlling access to documents and receiving and sending notifications based on CRUD (Create-Read-Update-Delete) operations. They are relatively inexpensive and lightweight. However, they are not powerful enough to support customized coordination requirements, such as monitoring deadlines, coordination of individual collaborators, ordering of steps, and they do not allow for integration with other tools. On the other extreme in the coordination spectrum, classic business process management systems, e.g. BPEL systems, are powerful and capable of supporting various forms of coordination. However, they are costly and complex to operate and maintain. In fact, these systems are too heavy-weight when considering the requirements of resource-centric collaborations as described above. A third family of solutions involves the development of special-purpose workflow-management systems from scratch, based on the particular collaboration task at hand. While a custom-made system can be tailored according to the requirements of the project, developing a

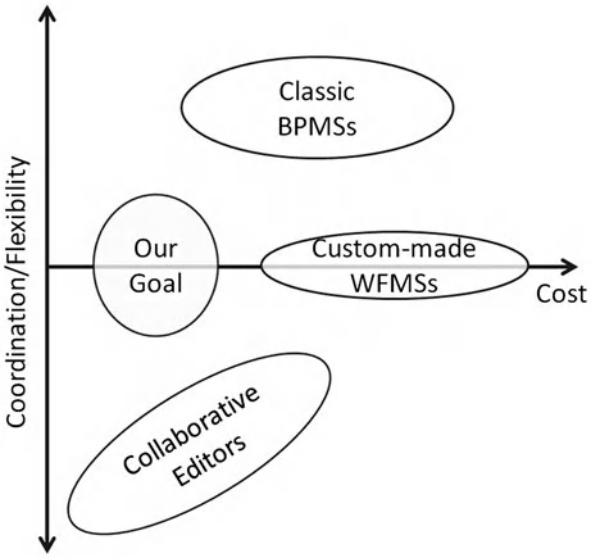


Fig. 9.1 Comparison of different solutions for supporting resource-centric collaborations

comprehensive workflow management system is far from trivial and substantially increases the effort required for supporting collaboration.

The problem of developing a systematic conceptual and software framework for supporting resource-centric collaboration is compelling, given the abundance of these activities. More specifically, our problem is to develop a "collaboration as a service" framework, through which to add sufficient coordination to a multi-user, multi-tool resource-processing environment on the web. In this paper, we describe exactly such a collaborating system that we have developed for supporting resource-centric collaborations, which properly balances flexibility and coordination needs.

The rest of the paper is organized as follows. Section 2 explores the related work. In Sect. 3, we briefly describe the language we designed for specifying collaboration activities. The architecture of our collaboration system is explained in Sect. 4. The methodology for integrating our collaboration-as-a-service system with other existing systems is outlined in Sect. 5. In Sect. 6, a sample project in which the collaboration system has been employed is provided. In Sect. 7, we present some possible future work and conclude with a summary and discussion.

9.2 Related Work

This work relates to several areas of software engineering including business process management, computer-supported cooperative work, and service-orientated software-engineering. This is why, this section is necessarily eclectic, and reviews

previous research that shares some basic characteristics with our conceptualization of resource-centric collaboration support. One of the key properties of resource-centric collaborations is the flexibility required in modeling and modifying the rules of the collaborative activity. Workflow flexibility is one of the major research topics in workflow management and has been studied for more than a decade [109]. There is substantial research on various aspects of workflow flexibility such as ad-hoc modification, flexible modeling, semi-structured workflows, and workflow adaptation. Burkhart and Loos [41], Schonenberg et al. [204], and Carlsen et al. [44] have surveyed different approaches for enhancing workflow flexibility and evaluated the support of workflow flexibility in a selection of workflow management systems. These approaches address the flexibility requirements of many types of workflows, but they do not provide a specific and comprehensive solution for resource-centric collaboration, as they do not intend to. However, many parts of this work were motivated by earlier research on workflow flexibility.

The artifact-centric approach [111] to business-process modeling shares some properties with our envisioned resource-centric collaborations. This approach focuses on the business data, named artifacts, manipulated and augmented during the life-cycle of workflows. There are various studies on artifact-centric workflows, e.g. [28, 87 166], but all of them share the idea of managing artifacts (which capture business goals) and developing services (which manipulate artifacts according to business rules). The two approaches share similar motivations and "artifacts" are akin to "resources". In fact, in our work, we use the term "resources" in a manner similar to "artifacts": in the REST sense of the term, "resources" are conceived as web-accessible repositories of "artifacts", whose XML representations are accessed through CRUD operations enable their clients to transition their internal behavior states. However, the artifact-centric approaches are deeply concerned with the internal structure of the artifacts, while our approach is agnostic of this structure and simply relies on the fact that resource representations are accessible through REST-style APIs. Furthermore, as artifact-centric approaches require a complete model of the structure of the involved artifacts, which may not be available, they do not provide the flexibility required by resource-centric collaborations.

Another related area of research is that of web-service orchestration, which focuses on the process of creating composite services by combining and coordinating a set of simpler web-services [188]. In the context of the traditional (SOAP-based) service-oriented paradigm, complex coordinated processes are typically supported by service orchestration. Process steps are performed through the invocation of web-service operations and the process logic is enacted by an orchestration system. The *Business Process Execution Language*, also known as BPEL, is the most popular web-service orchestration language [43]. There are plenty of engines, which support and execute orchestration specifications written in BPEL. Although BPEL is powerful, it does not properly support workflows that require intensive human interactions. BPEL4People [4] introduces an extension for BPEL to address human interactions as a first-class citizen. In fact, BPEL4People enables BPEL to integrate role-based human activities in orchestrations. While BPEL4People extends BPEL to support human participation, the combination of BPEL and BPEL4People is too complex

and rigid for supporting lightweight and flexible workflows such as resource-centric collaborations [58]. As a result, much research has been devoted to address this limitation. Some approaches adopt a RESTful approach [76] as the replacement for complex interaction protocols used in BPEL. Pautasso [180] proposed an extension for BPEL, named "BPEL for REST", which aims at enabling BPEL to support the native composition of REST services in addition to WSDL-based web-services. The same author also introduced a REST service composition system named JOpera [181]. Casati et al. [45] and Thone et al. [231] also developed models and systems for flexible service composition as alternatives to BPEL. Vrieze et al. [244] proposes a mash-up approach for supporting enterprise business processes. All the above-mentioned projects somewhat address the agility and flexibility requirements of the types of resource-centric collaborations we are interested in. However, they are not in accordance with human-driven nature of the collaborations. In addition, their language models do not provide first-level support for loose, non-deterministic ordering of steps.

Finally, we have to mention one more line of research related to collaborative workflows and resource-centric collaborations. Schuster et al. [206, 252] proposed a service-oriented approach and a system for supporting a type of collaborations, named "creative document collaboration", which shares many properties with resource-centric collaboration. Their approach mainly focuses on documents with compound internal structures, while our approach is conceived more generally to support any kind of resources. In addition, we believe that our coordination approach is more powerful than the rule-based model introduced in their work. Bite [58, 199] is a workflow-based composition model for web applications. It enables the creation of web-scale workflows through its lightweight and extensible composition language and engine. Bite can be considered as a genuine system for supporting collaborative processes as it addresses some of the main requirements of these processes such as web-human integration, lightweight process model, and flexible configuration. In fact, one of the Bite goals is to provide the support for collaborative workflows. However, Bite employs a sequential activity-based process model, which is not appropriate for modeling semi-structured workflows with loosely ordered steps, which is essential in the class of processes our work aims to support.

9.3 The Collaboration-Specification Language

The first step in supporting collaborations is modeling them. A collaboration model, a.k.a. collaboration specification, must define the elementary tasks that the collaborators may perform and the control structures through which these tasks will be coordinated. We studied various approaches and languages for modeling workflows. As we discussed in the previous section, there are numerous expressive coordination languages, which are not very appropriate for modeling resource-centric collaborations, as they are designed for structured workflows and support complex expressions for specifying a complete order among the process steps. On the other hand, there are

yet other languages, which consider flexibility, but do not provide the right amount of coordination for these collaborations and do not support the specification of "hooks" with external systems that may be used by the individual process steps. As a result, we decided to design a specific workflow language for supporting resource-centric collaborations, which balances between the required flexibility and coordination support. In the process of designing our workflow language, we took into account the properties and requirements of the collaborations in order to develop an intuitive workflow language, which is sufficiently expressive but not overly complex.[2]

The language is based on the event-driven paradigm, since resource-centric collaborations are human-driven and, as a result, the core coordination mechanism is responsible for monitoring and reacting to the actions of the users as they interact with existing interactive tools. At its core, the proposed coordination mechanism is *superimposed* [35, 116] on an existing ecosystem of tools, extending and regulating the functionalities of the existing tools with a coordination layer. At its weakest form, *expectative superimposition* does not affect the behavior of the basic layer; it simply "inspects" it. *Regulative superimposition* additionally allows the regulator to restrict the basic layer, by either delaying or blocking certain operations. Finally, *invasive superimposition* allows intrusive regulation, whereby the superimposed code is permitted to modify the underlying variables. Adopting this view in the context of an ecosystem of tools that support REST APIs, we develop a coordination service that combines expectative superimposition, to monitor the invocations of the underlying layer's REST APIs, and invasive superimposition, to invoke these APIs as required for coordination purposes. The coordination itself is specified in terms of ECA (Event-Condition-Action) rules. In these ECA rules, "events" are messages generated by external components (i.e., intercepted invocations of the REST APIs of the underlying systems used by the collaborating team members to perform their tasks) or other collaboration instances. "Conditions" are logical expressions regarding the state of the collaborative process. Finally, "actions" define behavior as the response to an expected event, under some conditions; these actions may be invocations of the underlying layer's REST APIs, or notifications forwarded to the tool users. The model of our collaboration language is illustrated in Fig. 9.2.

Types are data elements used for storing values in collaboration instances. They may be associated with methods for accessing and manipulating their values. The basic types supported in our language are *Boolean*, *Integer*, *String*, and *Time*. We also defined a special type named *User* in order to provide a first-level support for working with the data of the people involved. In addition, we included two collection types in the language, *Strings* and *Users*.

Functions are system-level methods used for managing collaborations instances and facilitating interactions. The supported functions include methods for *invoking web services*, *triggering events*, and *publishing errors*. Using these functions, a collaboration instance can communicate with other collaboration instances and external systems.

[2] The language is only partially described in this paper due to lack of space. The complete description will be available in Ghandehari's MSc thesis.

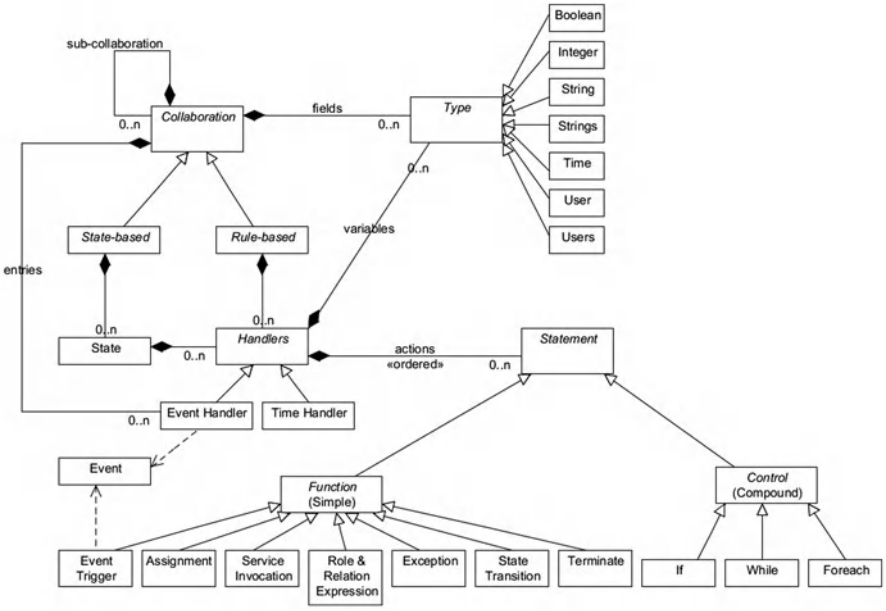


Fig. 9.2 The meta-model of our collaboration specification language

Control structures are used for defining the control flow among the collaboration steps, based on events and conditions. Control constructs supported in the language are: *sequence*, *selection*, and *repetition*. Sequencing is achieved through ordered execution of statements. Selection is supported by an *IfElse* statement and Repetition can be performed using *While* and *Foreach* statements.

In addition, the language supports collaboration composition and inter-process communication. Collaboration composition enhances the reusability by providing a mechanism for orchestrating a group of related processes and declaring a new complex process. Inter-process communication enables the interactions among different process instances at run time.

The language supports two styles for collaboration specification: state-based style and rule-based style. Each collaboration should be written in one of these styles. In the state-based style, one has to identify the states through which the collaboration instance passes during its life-cycle, and group the ECA rules based on the states to which they belong. In this style, ECA rules enable the transition of the collaboration instance from one state to another. In the rule-based style, there is no explicit concept of state or state transition; thus, a rule-based collaboration specification is roughly a set of ECA rules. Although these two styles support different means for organizing ECA rules, they are equivalent in terms of their expressiveness, and a specification written in one style can be transformed into the other. However, there is a significant difference in their applications. The state-based style best fits the collaborations that require more structured models as this style demands complete state-based behaviors

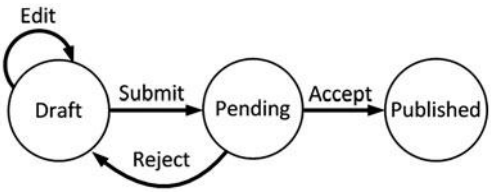


Fig. 9.3 The state chart diagram of the sample project report coauthoring collaboration

of the collaborations. On the other hand, the rule-based style is more natural for semi-structured collaborations as the style allows more flexibility in specifying the collaborations. Finally, it should be mentioned that one system can have collaboration specifications written in both of the styles, and the collaborations can interact with each other no matter in which styles they are written.

In addition to collaboration specifications, a configuration specification should be provided to the system. In this configuration specification, the elements used for interacting with the collaborations are defined. These elements include event definitions, pre-defined service calls, and methods for manipulating roles and relations. In a system, the configuration specification is shared among all collaborations.

Figure 9.3 depicts the state chart diagram of the project report coauthoring collaboration described previously in the paper. According to the diagram, the collaboration is initially in the "Draft" state, in which students collaboratively edit and update the report. Upon submission of the report, the state changes to "Pending", where the participants wait until the supervisor of the project decides about the quality and completion of the report. If the report is accepted, the collaboration proceeds to the "Published" state, which is the final state. Otherwise, the collaboration goes back to "Draft", so that the students may continue working on it.

In order to specify this collaboration example, we first need to write a configuration specification in which we define the events, roles, relations and services for interacting with the collaboration. In particular, events are the incoming channels by which external components send messages to collaborations; while roles, relations, and services are the means by which the collaboration responds to events or gets additional information. At the compile time, events are translated into methods on the REST API of the system, used by external components, which trigger events on relevant collaborations; while roles, relations, and services are compiled into system methods, used by collaborations, which make calls to external components. A sample configuration specification for this system is in Listing 9.1

Based on this configuration specification, we can now define our collaboration specifications. Written in the state-based style, the specification for our sample project-report coauthoring is provided in Listing 9.2. In this collaboration specification, we declare two fields for storing persistent data needed in the collaboration, i.e., the IDs of the project and report. The "Create" Entry, which works as a constructor, instantiates the collaboration and initializes the state and the fields of the newly created collaboration instance. The initial state, named "Draft", contains two event handlers: the "Edit" event handler only checks whether the person who triggered the event is a student and a member of the team; the "Submit" event-handler, in addition

```
// Event Definitions
Event Create (String projectID, String reportID);
Event Edit ();
Event Submit ();
Event Accept ();
Event Reject();

// Role Definitions
Role Student;
Role Professor;

// Relation Definitions
Relation Supervises(User supervisor, String projectID);
Relation Members(User user, String projectID);

// Service Definitions
String POST Lock (String reportID);
String POST Unlock (String reportID);
String POST Email (Users receivers, String content);
String POST Publish (String reportID);
```

Listing 9.1 The configuration specification for the sample project report coauthoring collaboration

to performing the authorization check, changes the state into Pending, informs the supervisor about the state change, and locks the report. The "Pending" state has two event handlers, both related to supervisor actions. If supervisor rejects the report by triggering the "Reject" event, the team is notified by email, the report is unlocked for further editing, and the collaboration returns to the "Draft" state. On the other hand, if the supervisor triggers the "Accept" event, it calls the "Publish" method, which finalizes the report, and the collaboration changes its state to "Published".

```
Collaboration StateBased ReportingCollaboration {
  // Field Declarations
  String projectID;
  String reportID;

  // Entry Specifications
  Entry Create {
    projectID = e.projectID;
    reportID = e.reportID;
    To(Draft);
  }

  // State Specifications
  State Draft {
    @Edit [Student]{
      If( ! (e.Sender Members projectID ) )
        Exception ("Permission Denied.");
    }
    @Submit [Student]{
      If( ! (e.Sender Supervises projectID) )
        Exception ("Permission Denied.");
      Lock(reportID);
      Email(supervisor, "Submitted");
      To(Pending);
    }
  }
  State Pending {
    @Accept[Professor] {
      If( ! (e.Sender Supervises projectID) )
        Exception ("Permission Denied.");
      Publish(reportID);
      To(Published);
    }
    @Reject[Professor] {
      If( ! (e.Sender Supervises projectID) )
        Exception ("Permission Denied.");
      UnLock(reportID);
      Email(team, "Rejected");
      To(Draft);
    }
  }
  Final State Published;
}
```

Listing 9.2 The specification of the sample project report coauthoring collaboration

```
Collaboration RuleBased DocumentCheckCollaboration {
  // Field Declarations
  Boolean TextChecked;
  Boolean FigureChecked;
  Boolean ReferenceChecked;

  // Entry Specifications
  Entry Start {
    TextChecked = False;
    FigureChecked = False;
    ReferenceChecked = False;
  }

  // Event-handler Specifications
  @TextCheck {
    TextChecked = True;
    If (TextChecked And FigureChecked And ReferenceChecked ) {
      Trigger(Checked());
    }
  }
  @FigureCheck {
    FigureChecked = True;
    If (TextChecked And FigureChecked And ReferenceChecked ) {
      Trigger(Checked());
    }
  }
  @ReferenceCheck {
    ReferenceChecked = True;
    If (TextChecked And FigureChecked And ReferenceChecked ) {
      Trigger(Checked());
    }
  }
}
```

Listing 9.3 The specification of the sample document check collaboration

Let us now consider a slightly more complex version of the project-report coauthoring collaboration. In this version, after the submission of a project report, a set of checks – i.e., text check, figures check, and references check – should be performed to ensure that the report is ready for the final review by the supervisor. There is no specific execution order among these checks, but it is important that all these checks are performed before the report is sent to the supervisor. In order to make this checking process reusable, we chose to implement it as a separate collaboration, named "DocumentCheckCollaboration". The specification of "DocumentCheckCollaboration" written in the rule-based style is presented in Listing 9.3. This collaboration has three fields to indicate whether the corresponding checks have been performed yet or not. The collaboration is instantiated and its fields are initialized when it receives the "Start" event. Upon completion of a check on the document, an event indicating the type of the completed check is sent to the collaboration that sets the value of the corresponding field to "True". When all fields are evaluated to "True", signifying that all the checks have been done, the collaboration triggers "Checked" event to inform its parent about the completion of the collaboration. In order to employ the "DocumentCheckCollaboration" in our "ReportingCollaboration", we first need to add the definition of the new events used in the "DocumentCheckCollaboration" in the configuration specification; then, we should modify the "ReportingCollaboration" collaboration specification to include "DocumentCheckCollaboration" as a "sub-collaboration".[3]

[3] Note that the underlying assumption in developing rule-based process specifications is that for each event type a single ECA rule is specified, thus eliminating the need for prioritizing or resolving conflicts among multiple applicable rules.

We used the state-based style for specifying the "ReportingCollaboration" and the rule-based style for specifying the "DocumentCheckCollaboration". The state-based style is more suitable for the "ReportingCollaboration", because there is a logical order among the steps of the collaboration; the corresponding states and their sequencing capture this logical dependency. Had we chosen the rule-based style, the states would have to be "simulated" using variables, which would result in a specification much more complex to write and to understand. On the other hand, the rule-based style is more natural for specifying the "DocumentCheckCollaboration", since the three checks can be performed in any arbitrary order. If we had chosen to write the "DocumentCheckCollaboration" in the state-based style, we would have needed to define nine different states to represent the possible combinations of the order in which the checks might be completed. Each of the states would have a couple of event-handlers but all of the event-handlers would almost do the same actions.

Clearly, these two styles for writing collaboration specifications have different usage scenarios and they complement each other. In our system, the collaboration editor supports both styles, so users can select which style they want to employ for writing each collaboration specification based on the properties of the collaboration.

9.4 The Software Framework

Having developed a language for specifying collaborations, the task becomes to develop a corresponding software system to support the specification and enactment of resource-centric collaborative activities. In fact, we are aiming to build a comprehensive tool-set to be integrated with multi-user, multi-tool resource-centric environments, in order to support resource-centric collaborations. In other words, we want to build neither a resource-processing environment nor tools to be used directly by end-users, but a behind-the-scenes supporting coordination-as-a-service system, which leverages the capabilities of existing resource-processing environments to support process awareness and coordinated user collaborations. The system consists of a set of software tools to specify collaborations and to manage their instances at run time, including the interaction of these collaboration instances with the users' activities, in the context of the existing systems. The main components of our collaboration-management system and their interactions are shown in Fig. 9.4 and are described in detail below.

The ***collaboration engine*** is the fundamental component of the collaboration system, which enacts the collaborations at run-time. The collaboration engine is responsible for: (a) instantiating collaboration instances; (b) delivering events to the collaboration instances; and (c) managing collaboration instances through their life-cycle. Once the configuration and collaboration specifications are written using the collaboration editor and compiled into executable collaboration specifications, they are deployed in the collaboration engine to be executed. When the engine receives an event requesting instantiation of a specific collaboration, the engine creates an instance and places it in the pool of active collaboration instances. The engine continuously

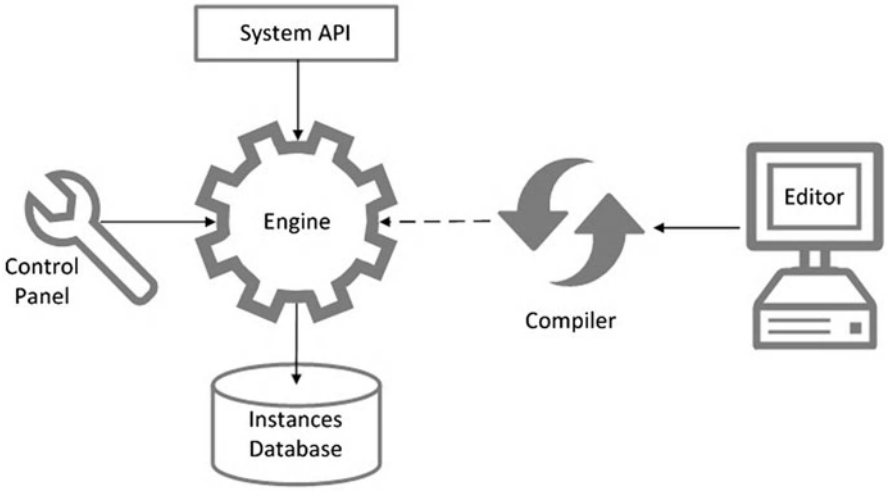


Fig. 9.4 The architecture of the software framework

listens for incoming events, whether from external systems or from active collaborations. Receiving an event, the engine triggers it on the corresponding collaboration instance, which causes the execution of the relevant event-handlers of the instance. If there is an event-handler defined in the collaboration for the incoming event type, and if the associated conditions are satisfied, the engine executes the enclosed actions, which may involve a state transition, sending a notification, or any other method call.

Interactions between the collaboration system and external components are possible through the system's REST ***API***. This API exposes the collaboration instances as web resources, which can be accessed and manipulated using HTTP methods. Receiving HTTP requests, the API transforms them into events and directs them to the corresponding collaboration instances. In addition, the API responds to the queries regarding the engine and the collaboration instances

The ***instances database*** hosts the active collaboration instances and archives the completed ones. This component is composed of a database management system and an access layer on top of it. Each collaboration instance is stored in the database as a set of records that include some generic metadata, such as ID and creation date of the instance, and some collaboration-specific data and their values. The engine uses the access layer to store and retrieve collaboration instances in/from the database. In addition, the access layer provides a caching mechanism for performance reasons.

The ***control panel*** supports system administration, including starting/stopping the engine, deploying specifications of collaborative processes, reviewing system logs, and configuring system parameters, such as the cache size, for example. In addition, it provides the ability for manually applying ad-hoc modifications on process instances in exceptional circumstances.

The ***editor*** is used for editing collaboration specifications. The editor is an *Eclipse*-based application, developed using the *Xtext*[4] framework. Given the grammar of our

[4] http://www.eclipse.org/Xtext/

language, Xtext creates an Eclipse-based editor for specifying collaboration in the language and builds a parser for the produced specifications. The resulting grammar-aware editor supports several interesting features, such as syntax coloring, code completion, code folding, and static error checking. Using this editor, users with not much programming knowledge, can easily specify a new, syntactically correct collaboration by following the editor's suggestions.

Since the collaboration specifications written in our language are not immediately understandable by our collaboration engine, it is required to transform them into executable specifications. To achieve that, we developed a ***compiler*** to go through the collaboration specifications, check their validity, and translate it to the target language. The input of the compiler is one configuration specification file, and one or more collaboration specifications; the output of the compiler is the corresponding collaboration implementations in Java language.

9.5 The Integration Model

One of our main goals in the design of the collaboration system was generality, so that the system can be employed in a variety of resource-centric environments. In fact, it should be possible to integrate the collaboration system with any arbitrary ecosystem of tools, in order to support the coordination necessary for their collaborative activities. For example, if we have a collaborative real-time editor, which provides an environment for sharing and co-editing documents but falls short in supporting the coordination requirements, we can integrate our system with the editor to employ the coordination capabilities of the system in the editor. To facilitate the integration process, we have developed a methodology for integrating the collaboration system within an ecosystem of other systems and tools, as depicted in 9.5. In this section we discuss at a high level model (and process) of integrating our system with an existing set of tools. In the next section, this will become clearer through the description of our case study.

The ***collaboration system*** refers to our toolkit as described in Sect. 4. The main responsibility of the system is to coordinate the users at run-time (using the collaboration engine) according to the rules specified using the editor.

The ***base system*** includes the users' resource-centric tools that need to be coordinated as part of the collaboration process. It is actually the target component, with which we want to integrate our collaboration-as-a-service system. Usually, the base system includes a set of editors and tools for working on shared resource repositories. In this work, we do not discuss how to develop a base system.

The ***base API*** mediates the interaction between the base system and the collaboration system. This API is actually an integrated interface into different elements of the base system; thus, calling one of its methods may result to calls to the editors, tools, or repository. Given the specific protocol according to, which the collaboration engine expects to communicate with the base system, it is unlikely that the base system directly supports the base API. In this case, we need to implement such an

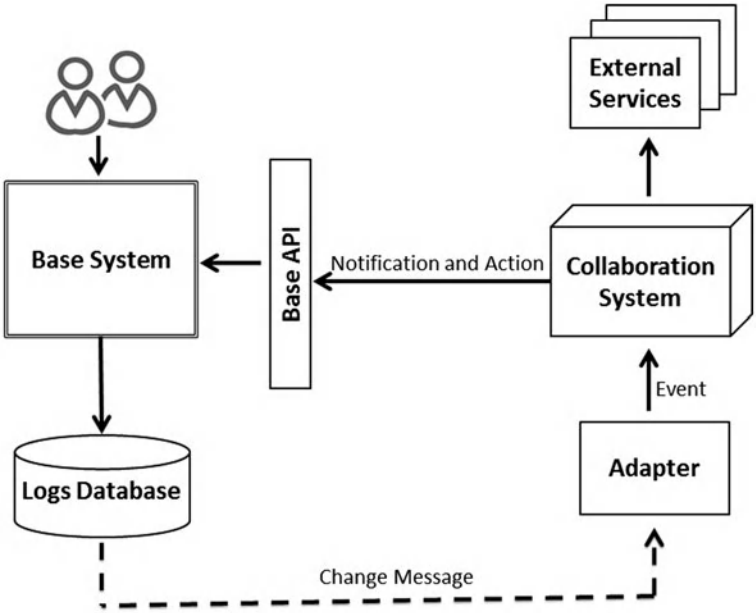


Fig. 9.5 Our suggested model for integrating the collaboration system with other tools

API as a part of the integration process. The set of methods required for the API depends on the events and actions expected from the collaboration system.

As we employ an event-driven approach for modeling and implementing collaborative activities, the collaboration engine needs to know about the actions performed in the base system. In fact, we need a mechanism to inform the engine about these actions. The ***logs database*** collects the logs of all the actions of interest, performed in the base system. The base system is responsible for populating this database; therefore, whenever an action is performed, the base system should generate and place the corresponding logs into the database. In addition, the logs database is responsible for publishing the changes; thus, it informs the listeners to the logs database whenever a new action log is added to the database. The ***adapter*** is responsible for transforming the action logs into events understandable by the collaboration engine. Effectively, this is the middleware implementing the connection between the base system and the coordination layer executing the collaborative specification. The adapter registers itself with the logs database and listens to log-change messages. Whenever a message is received, the adapter creates an event and sends it to the collaboration engine.

External services are used to extend the capabilities of the collaboration engine. External services can be called during the collaboration execution, using the service call function provided by the collaboration language. Therefore, if some more complex processing is required during the collaboration execution, which cannot be supported by the collaboration language, it can be implemented as a web service using any scripting or programming language, and then be invoked by the engine.

In addition, using external services is the mechanism for reusing already-available web services.

A typical scenario of interactions among the components above is as follows.

1. A user accesses a resource and starts working on it, using an editor, which is one of the tools of the base system.
2. Upon completion of the editing session, the editor updates the resource, which causes an action log to be added to the logs database.
3. The adapter inspecting the logs database recognizes a change message in the logs database, builds the corresponding event(s), and sends them to the collaboration engine.
4. The engine reacts to the event(s) by executing the relevant collaboration instances, which may involve calls to external services, such as for example to notify users through email, and invocations of the base-system APIs.
5. Invocations of the base API may result in further updates of the shared resources.

While designing the integration model, we focused primarily on ease of integration and low coupling between the workflow system and the base system. According to the superimposition model, the base system is not aware of the collaboration-as-a-service system, which monitors and coordinates the base-system activities. The base system only logs relevant actions with the coordinating-systems logs database. This may occur naturally; when the base system uses REST APIs to access a resource, these API invocations can be trapped to also inform the coordinating-system's logs database. If this assumption is not met by the base system, special-purpose "glue code" may have to be developed to accomplish this. The collaboration system listens to the log changes, with help of the adapter and acts upon them appropriately. Therefore, it is possible to change or replace the collaboration system completely without any need to modify the base system. In addition, the adapter makes the integration easy since we can change the base system without any need to modify the engine; but we only need to modify the adapter. Similarly, the collaboration system is only coupled to the base API; therefore, as long as the API is not changed, it is not important how each method of the API is implemented or if the internal structure of the base system changes, e.g. a tool is replaced or a new editor is added.

9.6 Case Study

The GRAND (*Graphics Animation and New Media*) Network of Centres of Excellence[5] is a multidisciplinary research network exploring the application and advancement of graphics, animation and new media in Canada. The GRAND Forum is a web-based application, designed to collect information about the people in the GRAND community, their relations, and their products and activities. The Forum has been implemented based on Mediawiki[6] framework, which is a web-based wiki software application, used for building various wikis such as Wikipedia.

[5] http://grand-nce.ca/

[6] http://www.mediawiki.org/wiki/MediaWiki

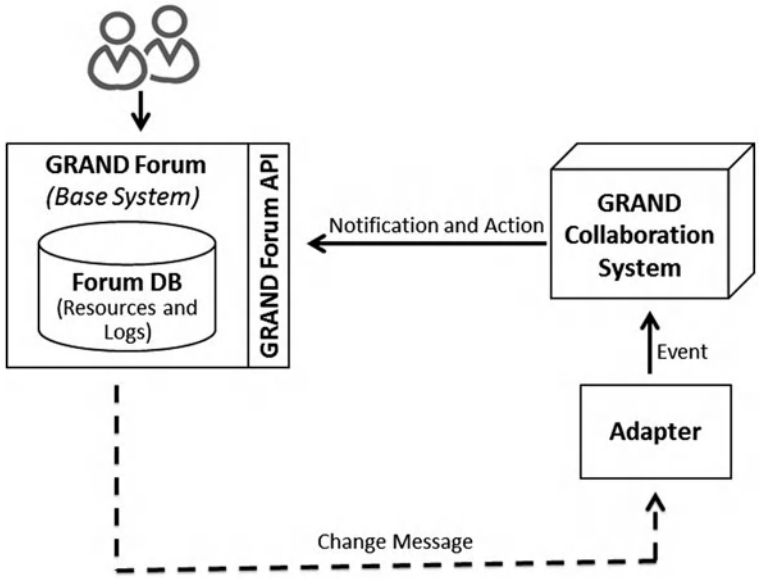


Fig. 9.6 The realized integration model for GRAND project

In the FORUM, it is possible to modify the underlying information resources using the provided HTML forms; however, this does not support the desired flexibility and the dynamics in performing the tasks. Many of the GRAND collaborations require interactions among various researchers, but the FORUM was not designed to support these types of interactions. For example, when a new researcher requests to join a project, the manager of the system and the leaders of the project should consent; however, it is not possible to support this collaboration completely in the FORUM as there is no way to ask the questions and act according to the responses.

After analyzing the collaborations needed in the FORUM, we found out that they share most of their properties and requirements with resource-centric collaborations. Therefore, we decided that our collaboration system can fit appropriately in the FORUM and provide the support for implementing the required collaborations. In order to employ the collaboration system in the FORUM, we tried to realize the integration model in the context of the FORUM. The resulted integration model is depicted in Fig. 9.6.

According to this figure, the FORUM is actually the base system to which we add collaboration support. In this context, the front-end of the FORUM acts as the editors and tools of the integration model. The FORUM has an underlying database management system integrated with Mediawiki. It is a MySQL database, which acts as the resource repository of our model and maintains all the data of the FORUM. It also hosts the logs database of the coordination engine; to that end, we created a special table, named Event table, in which the actions performed in the FORUM are logged. We modified the front-end so that when a user performs a collaboration-related action, typically submits a form, the front-end generates the corresponding action logs and add them to the Event table.

For publishing the actions logs, we employed the trigger mechanism of the MySQL database. Therefore, whenever a new record is added to the Event table, the defined trigger gets executed; consequently, the action log is sent to the adapter as a JSON message. Receiving a message, the adapter parses the message, creates an event and sends it to the REST API of the collaboration system. In practice, the adapter is responsible for transforming action logs form JSON messages created by the FORUM into URL-encoded events understandable by the collaboration engine.

Using an extension mechanism provided by Mediawiki, we developed a REST API into the FORUM that in fact is the realization of the Base API of the integration model. This API contains methods for accessing and updating entities and the relations of the FORUM; examples of these methods include methods for getting the information of a researcher, creating a new project, and adding a membership relation between a researcher and a project.

During the execution of GRAND collaborations, there are situations where some information should be provided to or asked from a user or a group of users; for example, the decision of a project leader on admitting or rejecting a new researcher. The FORUM already had a notification mechanism that we used for providing information to users; however, the FORUM did not provide the capability to dynamically collect data from users. To solve this problem, we extended the notification mechanism of the FORUM by developing special types of notifications that ask recipients for data. The basic type is ConfirmationNotification that asks a yes-or-no question from a user. In order to enable the collaborations to use the notification mechanism, we added a method on the REST API of the FORUM by which the collaboration engine can send notifications to users. When a user responds to a special notification, an action log containing the data collected from the response is added to the Event table that results in executing an event on the corresponding collaboration instance.

9.7 Conclusions and Future Work

In this work, we studied a prevalent type of collaborative work, namely resource-centric collaborations, in which a group of people collaborates in the development (i.e., creation, updating and deletion) of shared resources. We reviewed classic workflow-management systems, both aimed at enacting automated workflows as well as the ones designed to support collaborative resource manipulation, and we discussed their shortcomings. As a solution, we introduced our own collaboration-as-a-service system, for supporting this type of resource-centric collaborative work. We briefly explained the architecture of our system and the language it supports. In addition, we described the process for integrating our system with other base systems through which the participating users manipulate the underlying resources. Finally, we discussed a case study, in the context of which we validated our language and the corresponding resource-centric collaboration support system.

The main contributions of this work are our conceptualization of collaboration "as a service" and our the system we have developed to support collaboration in a

resource-centric environment. The approach is designed according to the requirements we identified for resource-centric collaborations, and the system enables the intuitive specification of the collaborations through its expressive collaboration language. The system is comprehensive in that it includes support for editing and enacting workflows; at the same time, it is light-weight enough to be used in resource-centric environment such as web-based systems. Finally, through the GRAND FORUM case study, we have demonstrated that our integration model is flexible enough to enables the deployment of our system in the context of any ecosystem of collaborative systems and tools.

There are several improvements that we envision for this work, specially its usability and extensibility. Currently, collaborations are written in a textual format using our Eclipse-based collaboration editor. Although the editor provides some support for authoring collaborations, a graphical editor would make the process much easier. We are working on a web-based graphical collaboration editor, which will be integrated in the control panel. During our work on the GRAND project, we discovered that collaborative projects usually have many similar collaborations such as membership management, reporting, and publishing. In addition, collaborations in a project or across various projects share some common collaboration fragments such as voting for distributed decision making. We believe that identification and classification of resource-centric collaborations and collaboration patterns will significantly contribute to this field. As the next step, the languages and the editors of collaborative systems can provide the support for these patterns, such as suggesting the patterns or facilitating their implementations. Finally, resource-centric collaborations typically have some steps that require asking questions from users and collecting the responses. In fact, this is the main method by which the collaborations can interact with human participants. This type of interaction almost exists in all of resource-centric collaborations we studied. For the GRAND project, we developed a notification mechanism by which the questions are sent to the users, and the responses are collected and informed to the collaboration engine. Based on the requirements of GRAND collaborations, we develop some general questions that can be used in any of GRAND collaborations. In fact, many of these questions exist in similar forms in various resource-centric collaboration project, for example yes-no question. Therefore, it is valuable to identify common questions, more generally common types of user interactions, and extends the collaboration language and system to support these questions/user interactions as first class citizens.

Acknowledgements This work was supported by the GRAND Network of Centres of Excellence, NSERC, AITF and IBM.

Chapter 10
Connecting the Dots: Using REST and Hypermedia to Publish Digital Content

Luis Cipriani and Luiz Rocha

10.1 Introduction

Media companies are very dependent on Content Management Systems (CMS) to accelerate and increase quality of generation, publication and distribution of content to its readers. This type of system is commonly found as open source or proprietary ready-to-use solutions, but our company decided to create one from scratch: the Alexandria Platform. This chapter presents a practical application of REST architectural constraints that is the basis of operation of Abril Mídia, and to share the lessons learned in 3 years of development.

The company is introduced in Sect. 10.2 as well as how its structure influences the chosen solution. Section 10.3 shows how the Platform is organized in technical terms. Section 10.4 presents how we applied each REST's architectural constraints. Section 10.5 is dedicated to the Uniform Interface constraint, because it has specific information about our interface. Section 10.6 shows some result analysis from the established goals. Then we wrap up everything with the conclusion in Sect. 10.7 and show the next steps to take with the Platform.

10.2 Abril Group

The Abril Group is one of the largest communications conglomerates in Latin America, it works with information, education and entertainment for a wide variety of customers segments, with integrated operations in several types of media. Founded

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

L. Cipriani (✉) · L. Rocha
Abril Midia Digital, Rua Sumidouro, 747 Pinheiros, 8th floor,
São Paulo 05428-070, Brasil
e-mail: lfcipriani@gmail.com

L. Rocha
e-mail: lsdrocha@gmail.com

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_10,

in 1950 and headquartered in São Paulo, Brazil, the Abril Group has more than 7000 employees, posting net revenues of BRL$ 3 billion in 2009. This Group is divided into four major areas:

- **Media**: produces content for magazines, their online version and other digital products;
- **Distribution & Logistics**: is a holding company responsible to distribute and sells the Abril Group publications;
- **Print Department**: the publications brought out by the Media and Education publishing houses are printed in this area;
- **Education**: offer a widely diversified Portfolio of more than 3000 titles consisting of textbooks and other teaching materials, collections and supplementary works.

The distributed system subject of this chapter was developed by the Media area, which encompasses the Editora Abril publishing house, MTV Brasil (television channel and Internet portal), Internet and Elemidia network company. Editora Abril is the first company in the Group to produce content for magazines and their online version. The Internet area develops products, content and services on a wide variety of platforms and in many different formats, responding to the expectations of consumers attuned to electronic media. It runs over more than 80 websites, in addition to more than 130 channels and services developed for mobile phones.

More details about the company can be found at the Institutional website [101, 102].

10.2.1 How Company Structure Influences the Solution?

> Organizations that design systems are constrained to produce designs that are copies of the communication structures of these organizations. - Conway's Law [38, 55]

As cited before, the media area generates content to be published no matter if it's printed or exposed in a digital environment. The process of managing and publishing content involves a set of activities, such as: writing, organizing, classifying, publishing, distributing, associating content, generating knowledge, reusing, etc.

A naive analysis could consider that creating a system that performs the tasks cited above is not so complex, given that several types of libraries, frameworks or proprietary solutions are available and ready to use. But when the non-technical or company-specific variables are added to the solution evaluation, the ready-to-use systems or libraries starts to be removed from the list of available options. Abril Mídia is a big area inside a big company, so it's very probable to expect a high diversity of several aspects intrinsic to the company structure that couldn't be ignored when deciding how to provide the solution that makes every relevant need met by the system specification.

These are important high diversity aspects we need to take into account for the decisions made by the product managers and software architects:

- **Business domains**: Abril Mídia has publishing houses that create content for several segments, such as: economics, politics, entertainment, fashion, tourism, cars, adult, lifestyle, children, science, women, art. These diverse segments have a huge impact in the list of requirements a system needs to meet, since each publishing house will have business domain specific requirements.
- **Business resources**: the publishing houses deals with a set of business resources (for example: people, venues, articles) that should have a common information structure between them, but sometimes that's not true. For example, the metadata that constitutes a Person to an economics magazine is different from a celebrity magazine.
- **Budget and Team size**: there is a large discrepancy in publishing houses budgets and team sizes that is related to how its business domain fits into company strategy or how profitable they are.
- **People and Culture**: given that we are dealing with a big company that works in several segments, we also need to deal with several kinds of people and work cultures.

As a premise, the system built to meet the requirements of content creation and distribution should make viable any initiative coming from the publishing houses, no matter how diverse are the aspects involved, given that this initiative fits to the company strategy and budget.

From these aspects we could extract some important architectural characteristics the proposed solution should have:

- **Modularity**: the system should group similar features/resources in a way that enables a specialized team or system to deal with it independently of the interactions with other modules;
- **Extensibility**: a system with this ability will have a low friction to include more features based on different needs from the stakeholders;
- **Independent evolvability**: a module should be capable to evolve without impacting the development of other modules;
- **Scalability**: different publishing houses will have different server loads, the system must be able to scale for different types of load models.
- **Uniformity**: the grow of the system should be guaranteed by controlling the way each module communicate the data that powers the creation of the company products.

The next section shows the Alexandria Platform, the distributed system created to meet all these characteristics.

10.3 Alexandria Platform

Abril Mídia has a long presence in the Brazilian Internet history, since 1996 it published from magazine websites to Internet products. Despite that long experience in the digital world, content management is always a big challenge a media company

needs to face, given the changes of platforms of publication, storage technologies, metadata, business evolution, user needs and so on. In the end of 2009, the company started to rethink the way content was being handled, because the legacy content management systems were breaking some of new digital initiatives. Then, the following goals for the new Content Management System were derived from the needs of stakeholders:

- The system should be capable of extracting greater value from the content produced;
- The system should accelerate the processes of managing and publishing content;
- The system should accelerate the creation of user interaction enabled products.

Combining these simple but meaningful goals with the architectural characteristics from the previous section, we show, in this section, how the Alexandria Platform was structured and implemented, and how this structure can empower the achievement of the stated goals.

Defining it technically, the **Alexandria Platform is a system of systems, distributed, decentralized but interconnected, that allows each component to evolve independently and the Platform to grow organically as we add support for more business needs**. We could divide each component of this distributed system into initial four categories (the number could increase with the evolution).

10.3.1 Domains

A Domain is responsible for managing and storing a set of resources that are related to a specific business domain. Clients use it to access and manipulate the resources, as shown in Table 10.1.

Each domain is an independently deployed system (usually web server + database + application server) that is exposed with an uniform interface (API) for client use. For example, to create a digital product for travelers we could use Editorial, Media, Annotation and Venues APIs.

The metadata that describe each resource is controlled and should be respected by the users to avoid inconsistencies and a consequent information lost due to incoherent data.

Other relevant point is that if a publishing house has a very specific resource to handle that is not supported by an existing domain, it could extend the functionality by creating a specific domain inside the product that could later evolve and one day be assigned as part of Alexandria Platform (the only premise is to “speak” the same language).

Table 10.1 List of Alexandria Domains and Resources

Domain	Domain Resources
Editorial	Articles, editorial lists, media galleries
Media	Images, videos
Annotation	Comments, ratings
People	People, professionals, celebrities
Venues	Restaurants, theaters, bars, stores, hotels
Attractions	Movies, plays, shows
Polls	Polls, Quizzes
Cultural contest	Cultural contests (questions and answers)

Table 10.2 List of Alexandria Services

Service	Description
Abril ID	End user identification system
Social Core	User behavior tracking, social graph
Notification	A service to enable email, toast[a], SMS and mobile notifications
Search	Enable business domain specific search
ICE	External content import tool
Clickcounter	Real time Portal homepages performance
MCP	Provides corporate employee identification for internal systems

[a] As defined by Wikipedia, Toast (in computing) is a small, informational window displayed by certain kinds of software, especially instant messaging clients. See http://en.wikipedia.org/wiki/Toast_(computing)

10.3.2 Services

The Services exist to manipulate and consume resources to enrich it before a product finally consumes it. Each service handle a specific business need and there is a high diversity of services. We can see the current available Services in the Table 10.2.

Each service is a independently deployed system (usually web server + application server, sometimes with a database) that is exposed with an uniform interface (API) for client use. A final product uses a Service the same way it uses a Domain with the goal of providing a better experience or a different context to end users.

10.3.3 Data Entry

This is the front-end system that journalists use to create content that later will be published on any digital media available in the Platform (see Fig. 10.1). Usually there is one Data Entry application for each Domain. Additionally, it must provide ways that accelerate the creation of content, such as to easily upload and edit images, classify the content and associate with other (types of) contents, control the state of an article (draft, available, etc.).

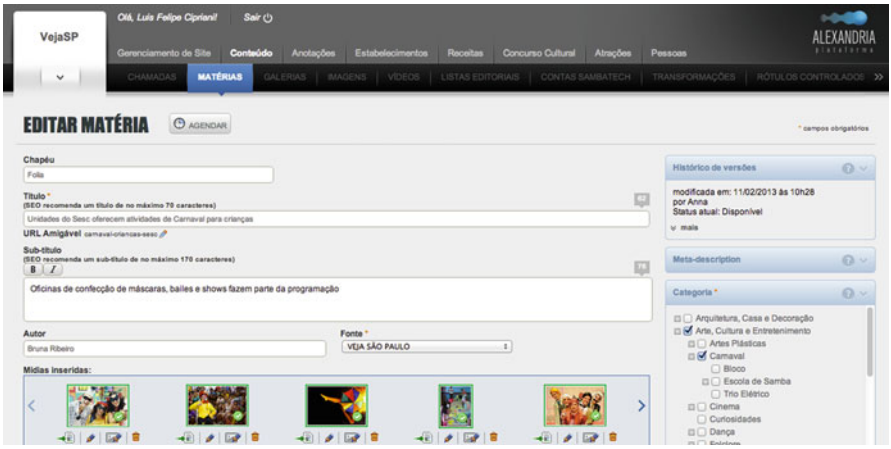


Fig. 10.1 Data entry: A screenshot of the form for editing an article, a resource handled by Editorial Domain. The bar on top shows links to other Data Entries or to managers of other Resources one Data Entry could handle

10.3.4 Site Tools

These are the set of tools the Webmasters of each publishing house that has a product published in the Alexandria Platform use to control how the website will look and how it will be structured. The tool does the association of content with the digital media it will be published. In a similar way as the Data entries, it must provide ways to accelerate the administration of a website, organize the home page, publishing of articles, scheduling of publications, etc.

The group of Data entries and Site tools is called Console (as showed in Fig. 10.1), that has the important role to empower the publishing house to create and publish content, and also provide ways to manage the performance of its digital products in qualitative and quantitative ways.

10.3.5 System Interactions

Alexandria is a decentralized platform. The lack of a central coordination authority allows each one of its systems to interact with any other. Some interactions are more frequent than others. The overall picture of the system use and interactions is:

- Domains and Services interact with each other to manage, transform, distribute and enrich content;
- Data entries input and manage content to Domains and Services;
- Site tools access, query and manipulate content to enable the creation of digital products for the company;

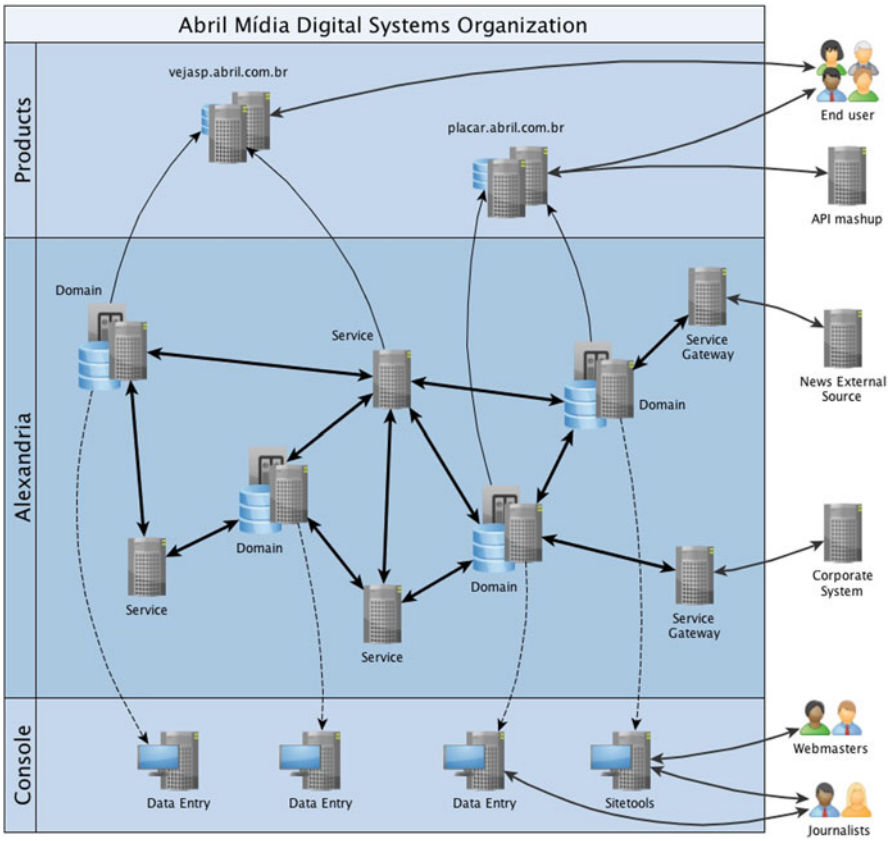


Fig. 10.2 High level representation of Abril Mídia Digital Systems Organization. Alexandria layer shows the system of systems architecture constituted of Domains and Services that interact with each other handling or enriching content (resources). All interactions inside this organization uses our defined uniform interface. (detailed in Sect. 10.5)

- Internal systems evaluate the current state of the business metrics and drive the next decisions aligned with the company strategy.
- There are some corporate systems (for example, Abril employee identification) that are used inside Alexandria through the implementation of Service Gateways. The same is done with External News Sources.

Figure 10.2 shows a diagram of each Alexandria component type and how they are connected to enable production and deliver of digital content.

The common case for Service-Domain interaction is the search engine pulling data from all available domains to generate indexes based on each resource and allow systems to query and find content by metadata, without the need to know the content identification or resource location. This is a key feature of the platform, products built on top of it make use of this ability to deliver webpages based on topics and other relevant metadata instead of being limited to human-managed content only.

To allow journalists to create media-rich content and establish relationship between articles, videos, images and other resources, Data entries interact with Domains to find and edit content, as well as validate any created relationship. It also interacts with Services such as authentication and authorization service to validate the user credentials before any action.

In the same fashion, the Site tools interacts with Domains and Services to allow webmasters to find and publish content. The Site tools then interacts with the products, pushing any definition set by the webmaster, enabling the product to pull data from Domains and Services.

The most frequently used Domains are the Editorial and Media Domains since all products built on top of Alexandria deliver in one way or another, media-rich news (articles with video or image galleries). This also explains why Search is the most used Service, as all of Alexandria products have a mechanism to allow end users to peruse in published news.

10.3.6 Operational and Technical Details

User access There are four types of users in Alexandria Platform and each one has a way to access and be identified in the systems:

- End users: represented by visitors in the websites powered by Alexandria. When a website offers exclusive content or functionality to these users, they use Abril ID service, which can be instantiated through a widget placed on the product page and in the future will provide also a OAuth[1] interface;
- Product system: represented by the product servers. They use a distributed authority identification scheme, in which some services must to know that a product will need to have access to it and all the authentication is done locally for them, without involving an external authority.
- Content creators: represented by journalists, graphic designers, reporters, editors, etc. They are identified through a corporate authentication system available from a gateway that exposes resources compliant with our uniform interface;
- Webmasters: the editors of the website structure. They are identified by the same way as content creators.

Authorization roles and permissions are implemented just for end users, content creators and webmasters. We do not enforce authorization schemes between systems to avoid implementation overheads that could decrease system performance. In this case, we rely on the security enforced by network standards.

In the Domains, the destructive operations (POST, PUT, DELETE) need to be authenticated by some user, so they need to check their authenticity in a central authority (the corporate one or Abril ID, depending on the type of the user).

[1] See http://oauth.net/

10.3.6.1 Platform Evolution

The process of adding a new component (Domain or Service) inside Alexandria involves to be compliant with the following constraints:

- The system must be compliant with all REST constraints;
- The system should implement and be adherent with our uniform interface. For a long time this constraint was verified manually, but an automatic validator is being created to ensure that no interface inconsistencies are added to the Platform;
- All interactions with other system must be done with applied principles of system robustness;
- The system should ensure that any destructive operation is being done with user or system identification;

Depending on the project, more constraints could apply, and the new set is defined by the technical or business roles of the Platform team. It is worth to mention that was really hard to enforce these restrictions on an environment that allow components to be included in a independent way and with several projects being developed in parallel. Still today we need to deal with some interface inconsistencies across several projects.

Once the system is created and deployed in production, changes are needed to be done in the products or other components of Alexandria to effectively use the new features, which means that the system coordination is manually maintained. Our experience shows that automatic system coordination isn't necessary (adding new components happens not so often) and could bring a set of risks that could decrease the performance of the Platform.

When the situation is adequate, there is the concern to extract a part of the system as a library that could be reused by our Platform and sometimes even by external users, through an open source license[2]. We don't think that the whole Platform could be reused by other company, since it has a set of very specific requirements and the priority was to deliver a good solution that could fasten the way the company publishes digital content.

10.3.6.2 Team Organization

Each Alexandria component or Product is a separate project, that could contain more than one code repository. A team is responsible by one or more projects. It is usual to group some similar projects in a concept, for example, Content Delivery solutions, User Interactions, and a team could be responsible for a whole group.

Every team has developers, a software architect, a project manager and a product manager. Sometimes a role could be acted by the same person across more than one team. Currently we have the following distribution in Alexandria Platform team[3]:

[2] See Abril Github page at https://github.com/abril

[3] These numbers have a high variation, they represent the situation at the time of writing this chapter.

- 50 Software Engineers (back-end, front-end and test developers)
- 12 Software Architects
- 12 Information Architects (user experience architects and graphic designers)
- 6 Project Managers
- 4 Product Managers
- 1 Platform Advocate

They are responsible of:

- 8 Domains
- 12 Services
- 14 End user Products

A large growth is expected in the number of End user Products (Abril has at least 80 websites) as they are migrated from the legacy CMS to Alexandria. The growth in the number of domains and services is stagnating and should not increase significantly in the next years, because all the important types of contents are already present. Also a growth is expected in the complexity of the End user Products, Services and Data Entries as they evolve from a business perspective to a scenario that increases user interaction, real time content delivering and knowledge generation.

10.3.6.3 Technology Applied

The most frequent technologies employed on Alexandria are:

- Languages: Ruby and Java (50 % each)
- Application protocol: HTTP 1.1
- Storage systems: MongoDB, MySQL, HBase, HDFS, PostgreSQL, Redis, Memcached
- Web servers: Apache, Nginx, Passenger, Jetty
- Web frameworks: Rails, Sinatra, Play, Goliath, Jersey
- Other tools: Solr, Hadoop, RabbitMQ, Varnish, New Relic
- Application deployed in virtual and physical environments, located in our own Data Center or Third-part Services (AWS, Heroku)
- Private Cloud for development environments

The reason that we choose between Ruby and Java as official languages is simply because before Alexandria development was started, Abril had several Java developers and projects in production. The Ruby influence came from a startup called Webco that was acquired by Abril in 2009. We had experience with other languages in non-critical projects, but Ruby and Java naturally became the main choices.

Given that each Domain and Service is a different system deployed and everything communicates with an uniform interface, we have the freedom to choose the most adequate technology to solve a specific problem. However, it's important to keep focus because if happens to have a high diversification of tools employed, it starts to get really difficult to hire specialized people to take care of all these technologies.

Table 10.3 Instances of Separation of Concerns

Client	Server
Website: use the resources to create a specific experience to reader	**Domain**: manage and store the resources
Data entry: creates a better user experience for journalists to create and manipulate resources	**Domain**:manage and store the resources
Domain: uses services to create relationships between resources	**Service**: provides specific data that enriches some resources

The inherent independence existent between each system brings some great advantages, such as:

- It eases problem solving, once it's all isolated;
- Gives freedom of technology choice, a characteristic that is valuable for good developers;
- Enable decentralized project management;
- It eases incremental adoption of development process improvements;
- Narrow focus when defining product roadmap.

10.4 REST Constraints Applied

This section shows how we applied each of REST architectural constraints proposed by Roy Fielding in his Dissertation, "Architectural Styles and the Design of Network-based Software Architectures" [76], in our Platform. In Sect. 5.1.8 of his Dissertation, Fielding shows a summary of the style derivation that constitutes what REST is, and the proposal is that REST is: **REST = LCODC$SS + U**

The formula above can be read as: REST is a architectural style that is derived from Layered Code-on-Demand Client Cache Stateless Server plus a Uniform Interface. This section will show just the LCODC$SS part, and Uniform Interface will be detailed in Sect. 10.5.

10.4.1 Client-Server (CS)

This architectural style is very often encountered in distributed systems and it's not an exception to Alexandria Platform, in which all applications deployed follows this style. Fielding [76] identifies three properties that influences Client-Server style and by so, influences how our system behaves.

Separation of concerns is achieved by separating the responsibilities between the Alexandria Domains, Services, Data entries, Site tools and Products created by the publishing houses. The Table 10.3 shows some instances of separation of concern property that occur in the Platform.

It's very important to take care of the responsibilities of each system, because we found it's easy to break the separation of concerns especially when teams are being pressured to deliver features fast and do this by relaxing the architectural constraints.

Sometimes we could find components, such as Alexandria Domains, that act as clients in some situations, and as server in others. This is common for Alexandria and didn't generate any problem, since each instance of separation of concerns behaves differently and don't share concerns.

By separating functionalities, we simplify the interface and force the client to decide how to use that set. This simplification decreases the risks and complexity, achieving **scalability**. For example, Domains only deals with resources and this brings simplicity when we need to scale our system.

Independent evolution is the only property that we couldn't guarantee in 100 % of the Platform, because it depends strongly on interface consistency across all components. Some of the reasons we believe that we couldn't keep interface consistency are:

- Initial uncertainty of the set of responsibilities a component should have;
- Teams working independently without common knowledge sharing;
- Known tech debits that were not prioritized by the Platform strategy, sometimes due to pressure for delivery.

To keep interface consistency is probably one of the biggest challenges when you deal with a large and diverse Platform, with teams working independently and pressure to deliver fast. Is highly recommended to prioritize the creation and control of Platform interface.

10.4.2 Stateless (S)

Since the beginning of the architecture design, the Alexandria Platform was implemented to be as compliant as possible with this constraint, because **scalability** is one of the main non-functional requirements that should be present on the system. Furthermore, stateless architectural style is a consolidated practice in teams that work with web applications or back end distributed systems, so even when, accidentally, the software engineer forgot to apply this constraint by design the problem got solved indirectly by the frameworks or libraries used to implement.

However, we needed to break this constraint in some situations by maintaining some state for destructive operations in resources, in order to identify the user of the service that's doing it. This was implemented with a cookie that represented an opened session in the Data entry. Despite the fact that this is a technical debit that could be solved with other authentication techniques, we decided to assume what we consider a low risk, because only affects destructive operations.

There are other important observations about the properties of this architectural constraint:

- **Visibility**: to keep the interface visible to components, beyond the need of not maintaining any state between the requests and responses, we also avoided to use SSL in the Platform and assume that we need to control the systems (internally or externally) that have access to the network.
- **Reliability**: not maintaining state eases the problem solving because we can isolate the failures.
- **Network performance**: despite the fact repetitive information is sent in the requests to implement stateless architectural style and this makes the network performance decrease, by far we assume that this risk isn't critical to the overall performance of the Platform.

10.4.3 Cache ($)

This architectural style brings great benefits to **network efficiency**, **scalability** and the **performance perceived by the user**, however it demands an efficient caching strategy to guarantee that the user never gets a stale resource, that can decrease the **reliability** of the system.

At Alexandria Platform we use HTTP as the application protocol, so we have a great availability of caching tools ready to use (we largely adopt Varnish, as shared caches), but one cannot assume this as an applied constraint until each API has a caching strategy implemented and tested. The main points that needed to be defined when implementing a good caching strategy are:

- What need to be cached;
- How long each resource should be fresh to use;
- What to do when you have a stale resource.

We faced some problems with caching and very often the reason was an inexistent or inefficient caching strategy, especially because the understanding that caching is an optimization technique and could be taken into account in later phases of the project combined with the prioritization of the product backlog. Given this, we share some lessons learned when dealing with caching strategies:

- Caching strategy could be created earlier, even if based on few assumptions about the resource time-to-live;
- As your system and the user load model changes, your caching strategy needs to evolve together;
- Avoid the need of cache purge, this breaks the client-server constraint, introduces a global coordination problem and the system won't scale. Try to rely only in your time-to-live strategy.
- Don't abuse of shared caches, sometimes is better to have one caching tier per client-server communication to ease solving of problems and scalability.
- Try other cache locations such as Local Caches to shorten the path between the request and response for heavy loaded services.

10.4.4 Layered System (L)

As we need to maintain a large distributed system that has some external dependencies, the Layered System architectural style is crucial to **encapsulate complexity** (bringing simplicity to the user) and allow an **independent evolution** of the components. Even with the decrease of **user perceived performance**, by adding layers of communication, we also assume that this is a low risk we take, given the great benefits we obtain.

Beyond the obvious instances of this architectural style, such as shared caches and load balancers, that is also a very consolidated practice in distributed systems focused to deliver digital products, the Alexandria Platform used this style to solve some critical integration problems, such as:

- **Encapsulation of corporate employee identification**: at Abril, the system responsible to identify the employee is implemented with proprietary technologies and uses another interface. To solve this, we implemented a gateway to this system that exposes an interface that is compliant with Alexandria Platform directives.
- **Encapsulation of legacy systems**: the old publication systems and legacy end user identification systems were encapsulated with gateways that expose the required uniform interface. Another benefit is that we could evolve the legacy systems incrementally to the current uniform interface without stopping to use it, it's just a matter of disable the legacy gateway feature API entry point as we activate the newly implemented feature API entry point.
- **Gateways to expose the APIs externally**: for information security issues, all Alexandria Platform network is controlled and used only internally, but there are requirements that forces the architecture to have points of exposure of this API to external users, for example, to build web widgets or to enable company providers to create products on top of the Platform. Expose an API implies to take care of user rate limits, enhanced access security, better caching strategies, so the Layered System style has an important influence in the implementation.

10.4.5 Code-on-Demand (COD)

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This **simplifies** clients by reducing the numbers of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system **extensibility**. However, it also reduces **visibility**, and thus is only an optional constraint within REST [76].

Alexandria Platform applies this constraint only for the Data entries, which are rich client applications where the journalists create content and Webmasters manage the website. The code-on-demand comes in the form of reusable JavaScript widgets that encapsulate a common action, for example, creating relationships between contents or uploading a media (images, videos, etc.).

With the premise that we (providers of Alexandria Platform) control the metadata, Code-on-demand could also be used to generate Data entries based on a domain specific language that defines the content metadata and how to handle it. But this scenario is too advanced for the current architecture and still not crucial for the needs of the Platform.

10.5 Uniform Interface

Uniform Interfaces, as proposed by Fielding [76] in his thesis, need four constraints to be introduced in order to achieve the expected architectural behavior from its components:

- Self-descriptive messages
- Identification of resources
- Manipulation of resources through representations
- Hypermedia as the engine of application state (HATEOAS)

While necessary to create a uniform interface to system resources, this set of constraints is not sufficient to fully design the application programmable interfaces (API) the Alexandria Platform need. For that, we're required to map business needs and translate it to a set of guidelines to describe and manipulate resources too.

10.5.1 Resources

The first thing to do is set up a set of constraints to allow resources to be uniquely identifiable. In the Alexandria Platform, a resource must have both of the following properties:

- **Unique Domain ID**: every resource must have an id, and this id must be unique in the Domain that the resource belongs.
- **Resource type**: every resource must explicitly declare its type. Resource type is closely tied with the Domain responsible for maintaining it.

Every Alexandria resource has a unique Domain id and an explicit type, no matter how simple or complex a resource is, it must have these properties implemented. This way, we can link to a resource anywhere in the Platform knowing that we'll reach the desired resource. This also creates a clear distinction between an Alexandria resource and a simple, structure data used elsewhere.

Table 10.4 Expected Properties of a Content Resource

Property	Purpose
Brand	Identify the magazine and/or business unit that "owns" the Content
Source	The source of the Content
Slug	A more human-friendly way to address the Content
Status	Whether the Content is available to consumption or not

10.5.1.1 Abstract Resources

We also selected a few common properties a resource could have and bundled those properties in a concept we called Abstract Resources. One of those Abstract Resources is a Content. Naming those specific set of properties proved to be useful, helping organize and create a shared understanding of things. Table 10.4 lists the expected properties of a Content resource.

Content properties can be extended. For instance, if a new Content needs more than the two statuses defined, it can add their own, as long as it keeps the base statuses untouched.

We applied the same pattern of picking a few significant properties and bundling it in named resources to abstract away a few other concepts such as:

- **Auditable Content**: keep basic audit data
- **Related Content**: list of Contents related to a given resource
- **Classified Content**: lists of attributes that classify a resource
- **Geotagged Content**: adds geolocation properties to a resource

Breaking down desired properties in clear abstractions allowed actual, concrete resources to be assembled by composition. It also helps organizing desired resources properties in coherent packages, which makes understanding straightforward.

It is an important business constraint too, as it makes it possible for Alexandria to support a large range of resources, sharing common properties and yet separate whatever the business considers to be different.

But most of all, separating desired properties in a generic, composable fashion allows for a better media type design and makes implementing representations easier. Each building blocks has a clear purpose and can be independently evolved. The ability to glue together pieces of domain-specific data and properties to a group of well-known properties is key to building a hypermedia-driven architecture, as it provides a clear way to build self-descriptive messages.

That said, it was our desire to avoid to classic pitfalls, the HTML soup tag and the XML Schema proliferation. Having no shared understanding or allowing resources to implement the same set of properties in incompatible fashion was out of the question, as it would raise the costs of the Platform and generate much more noise in the development. Having too much extensibility would incur in a increasing complexity to maintain media types.

10.5.1.2 Concrete Resources

Once we defined Abstract Resources and made them composable, creating our Concrete Resources is a matter of understanding the business needs and modeling the resource specific properties accordingly.

To understand how Concrete Resources are built, lets analyze the most common resource in use. All digital products currently built on top of Alexandria uses Articles in some fashion. Articles have properties like author, title, caption, and article body. An Article is a Content too and, as such, inherits its properties. Therefore, an Article resource looks like this:

- **Article**: author, title (or headline), caption, article body
- **Content (inherited)**: brand, source, slug, status
- **Resource (inherited)**: unique id and resource type

Actually, this is just a constricted example. An Article also inherits from Classified Content, Auditable Content and might have a list of Related Content as well, which makes it a little more detailed and complex.

Concrete Resource may share properties, attributes and even be composed with the same abstract resources and still be different business objects, as long as they have different resource types and each instance is uniquely identified.

10.5.2 Resource Identification

Alexandria resources are uniquely identified in the whole platform by its unique Domain id and resource type, this properties also reflect in the choice for a platform Relative URI Reference [26] convention:

This approach has some advantages; it provides a simple rule to build URI references for resources, making it easier to implement them, it also introduces a little predictability, which helps developers and testers during the development stages of the Platform.

But the main benefit expected from this convention was take further advantage of the layered nature of the Platform architecture and let each layer represent a given resource according to its purpose. For instance, the domain serving a Venue resource will deliver it using the appropriate representation (more on that in Sect. 10.5.3) while a website built on top of Alexandria, will render the same resource as a web page. In both cases, the Relative URI Reference stays the same—still the same resource—but the host (and thus, the Absolute URI) handling the request differs, as in one case we have a domain serving the resource and in the other, a website.

On the other hand, this approach limited the amount of customizations business analysts, webmasters and editors could do to the website URIs, in order to leverage off-the-shelf Web metrics applications, SEO practices and all sorts of tools that use website URLs to extract business intelligence. And while it did not hamper development and operation of any digital product, it proved to be a questionable trade-off.

10.5.3 Representations

The media type of choice for the Alexandria Platform is JSON [57]. It has been widely used in the web since 2004, has built-in support in almost all mainstream programming languages, has more than enough tools available to edit and visualize and is perfectly human-readable, which is an asset during development.

The other options at early stages were using Atom [172] and AtomPub [98]. They seemed a better option at first, especially to handle Articles, Comments and other editorial content, but as we built the Platform, JSON proved to be more flexible.

Representations also have to be self-descriptive. A tendency in REST architectures is believe this Uniform Interface constraint should only be applied to messages and not the data being moved in the network. But designing representations to be self-descriptive is key to leverage independent evolvability and the longevity promoted by the architecture, making it future-compatible and robust. Specially when using a free-format like JSON.

As an example, when implementing the Venue resource, the first proposal made by the development team was to list each single service provided by the Venue as a key in the document root, something like this:

```
{
  "service_1": "service 1 name",
  "service_2": "service 2 name",
  "service_2_notes": "general notes regarding service 2"
}
```

While perfectly valid JSON, this implies that the client will have to have previous understanding of the semantics to make sense of the data. There is too much coupling and brittleness in this as it requires clients to evolve with the server, when the latter introduces changes to the Venue representation.

A better approach, later proposed by the same team is:

```
{
  "services": [
    {
      "id": "service_1",
      "name": "service 1"
    },
    {
      "id": "service_2",
      "name": "service 2",
      "notes": "service 2 notes"
    }
  ]
}
```

While the second version of the representation is definitely more verbose, it's also more self-descriptive, has enough metadata to allow clients to search for what they

need and the introduction/removal of elements in each element does not break clients that do not understand the new semantics.

Again, this is a basic example. But we found out that badly designed representations can and will introduce coupling, limit visibility and hinder the ability of the platform to evolve independently.

10.5.4 Hypermedia

At the beginning of the Alexandria platform development, no JSON hypermedia link standard [170] had emerged or was being actively advertised, therefore, we rolled out our own format for JSON linking. Obviously, as soon as a standard emerges, we'll take the time to update our representations. For now, our JSON hyperlinks look like this:

```
{
  "links": [
    {
      "href": "http://editorial.api/articles/id-1",
      "type": "application/json",
      "rel":  "article"
    },
    {
      "href": "http://venue.api/venues/id-1",
      "type": "application/json",
      "rel":  "venue"
    }
  ]
}
```

This format has been in production since the launch of the platform and worked out pretty well so far, but suffers from one problem. It advertises only one media type per link, which is restrictive. This creates coupling as Clients might use this as an authoritative information instead of passing what they can handle using an Accept Header. On the bright side, it uses the resource type as a Link Relation [171] and lets the Client pick whatever resource it wants to navigate to. This has proven to be a nice engineering decision. It also implements the self-descriptive approach described on the previous section, making it possible to add and remove a number of links without breaking clients, as this is not a coupling point.

In some cases, a bit more metadata is introduced in the link structure to allow more complex decisions when navigating between resources and to reduce unnecessary requests. The most common case is, the web editor only wants to render the links of resource in a "related content" box in the website page. To avoid extra requests to each linked content just to retrieve its title or headline, we embed this metadata inside the link:

```
{
  "links": [
    {
      "href": "http://editorial.api/articles/id-1",
      "type": "application/json",
      "rel":  "article",
      "title": "sample article",
      "preview": "http://medias.api/media/id-314.jpg"
    }
  ]
}
```

This has an obvious trade-off; if the headline of the linked article happens to change, the metadata in the link will not be updated, so there's a good chance of "metadata rot" in doing this. The business stakeholders accepted this risk, but perhaps the best approach for a situation like this is to implement a Partial GET, allowing the client to specify what it wants in a query string like this:

```
GET   /articles/id-1?fields=headline,preview HTTP/1.1
Host: editorial.api
```

This approach is good because it keeps consistency and leverages the Cache constraint of a RESTful architecture.

10.6 Evaluation

In Sect. 10.2.1 we mentioned four important architectural characteristics that should be addressed: Modularity, Extensibility, Scalability, Uniformity. So let's evaluate how each aspect was implemented:

Modularity is achieved with Domains and Services, each component entails a partition of the set of features Alexandria Platform offer to its users. A relevant trade-off that appeared by the increasing number of modules was the impact on the operation team that needed to adapt fast to be able to handle and monitor all the new components. **Extensibility** is enabled by the independence between all Platform components and the adoption of a standard uniform interface; then one team can build an extension inside a new component and all will be integrated with the uniform interface. **Scalability** comes from the stateless nature of the components (which is also a REST constraint) and its inherent simplicity; we also take care of adopting technologies that have support for this requirement. **Uniformity** was shown in details in Sect. 10.5, but even with all the established definitions, this aspect was the hardest to ensure because we had several teams working in parallel when it was being defined; so it's a common sense for us that the uniform interface will change to evolve as the Platform become more mature.

Section 10.3 introduced the company goals for digital content. They are more related to the functional aspects of the Platform and the current situation is the following:

- *The system should be capable of extracting greater value from the content produced*: we could consider that this goal is being reached gradually, since the majority of our current work is still focused to deliver a more reliable platform and to do maintenance in the production deployed systems. We didn't explore the full potential of extracting and enriching content to generate knowledge.
- *The system should accelerate the processes of managing and publishing content*: is achieved by the work User Experience team have done in the Data Entries; journalists and webmasters are gaining confidence gradually with the new interface and its evolutions;
- *The system should accelerate the creation of user interaction enabled products*: Annotation Domain and Services such Social Core, Abril ID, Notifications are the solutions available to address this goal.

From the development process point of view, despite the fact that the platform was built to make agile delivery of new products, the experienced time to deliver was slower than the time used for projects that used legacy systems. However this is easy explained by the fact that the new Platform enforces constraints that were never took into account before, such as uniform interface, robustness and performance. So we are taking longer time, but the quality is way better than before.

From a business and cultural perspective, we faced some problems to convince corporate users to change their mindset to a new publishing platform, given that the legacy publishing system is much more flexible, in an aspect that they have all the freedom to do whatever they want to (just) deliver content, but any other initiative such as, extracting value and knowledge, expose an API, share content between publishing houses would be broke by the legacy system architecture. We needed to do a lot of meetings and corporate trainings to be able to change the way Alexandria concepts and goals were understood.

All websites powered by Alexandria (at the time of writing), from January 15th 2013 to February 14th 2013, delivered 12.5 million pageviews. The most used domains, Editorial Domain (that serves articles) has an average throughput of 4130 requests/minute; Midia Domain (servers images, videos) has an average throughput of 4730 requests/minute. We consider our infrastructure stable.

From a perspective of how the concepts of REST are understood by developers, the same frustration cited by Roy Fielding in the blog post "REST APIs must be hypertext-driven" [79] applies to us. This misunderstand that happens with REST principles was a factor that indirectly influenced the teams to make some wrong design decisions in the code base. Our experience showed to us that spending some time explaining and coaching teams about the principles could eliminate some inconsistencies, developer frustrations and doubts about the real advantages of doing the system with that constraints.

10.7 Conclusion

After 3 years developing the Platform and digital products on top of it, we can pretty much state that the promise of extensibility, uniformity, modularity, scalability and independent evolvability associated with adopting REST as the Platform architectural style has paid off.

We made our fair share of mistakes and bad decisions. A few driven by business pressure and a few from bad design or engineering decisions. But overall, the properties we wanted and expected were realized and, with that we have a resilient enough Platform to evolve together as our business strategy evolves.

Some of the bad choices will get fixed eventually and some will not and the Platform will live with it. It is impossible to have a picture perfect architecture and still implement all that the business needs and desires. Most, if not all, of our trade-off are well known, though, which makes working with this decisions easier.

10.7.1 Lessons Learned

There was a lot to be learned building the Alexandria and there will be a lot more to in keeping it up to date for years to come, specially considering how quick the digital media business changes. Below is a list of what we learned in this first years of Alexandria and consider the most valuable lessons so far.

10.7.1.1 Robustness Over Protocol Optimization

The protocol is the set of rules and guidelines that determine how distinct parties communicate back and forth. On a system of systems platform like Alexandria, which consists fundamentally by heterogeneous parties communicating back and forth, the protocol is of utmost importance. And in a platform where the heterogeneous parties are encouraged to evolve independently from each other, the chosen protocol must be flexible and robust.

We learned that picking flexible protocols have trade-offs on its own. Robustness and flexibility comes with a price that forbids local optimization of the protocol. To allow a protocol to favor a use case or system will, invariably, hurt all other systems. In some sense one can argue that flexibility and robustness is, itself, an optimized state.

The truth is that trying to preview what changes and business needs the platform will be supporting is an exercise in futility. Digital media is a shifting business and the platform has to be able to support it for as long as it is in production. Alexandria must be future-proof.

One of the choices we did regarding protocol robustness was picking JSON instead of Atom and AtomPub as the Media Type of the Alexandria. While there's nothing absolutely wrong with Atom, it was a great fit for our need at the time, which was

publishing articles and associated media. But exercising a little creativity, we saw that if we needed to evolve it a little bit to support future Alexandria needs, like Venues and Notifications for example, it would require us much more effort to fit it in Atom. Atom was optimal for basic content, but it would be a bad fit for other things we believed we'd have to implement.

On the other hand, JSON wasn't a great fit for our (at the time) need. But its loose format would allow us to model it in any shape we wanted, making it a great future-proof choice, even if we had to develop some tools that Atom already had.

REST is a long term bet We believe that the duration of our effort and the time spent discussing and elaborating specifications proved to us the trueness of the following Roy Fielding quote in one of the comments of a blog post: "REST is software design on the scale of decades: every detail is intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency." [79]

It is still hard for us to enforce REST constraints for projects that should be delivered in short-time and has lower relevancy for the company. The most difficult aspect to be enforced is the Uniform Interface that is frequently stated as implementation overhead. The drawback generated by systems that don't respect the interface is incoherence and decrease of integrability of the systems. The interesting is that in some cases, assuming these risks isn't a bad idea and a later refactor of the project after gaining more business maturity is a better way of dealing with this lesson.

We consider that our scenario fits very well in the REST constraints and the initial assumptions of the process of building the whole platform never considered a short-time to execute. Therefore, we placed the bet and we yet haven't any critical reason to think we are losing.

The protocol interface and metadata must be kept sane Sometimes, in order to deliver something fast or to avoid what was perceived at the time as over-engineering, we are compelled to abuse or introduce small but significant changes to our metadata or the interface. From a single system viewpoint, stuff like that doesn't look like a major crime. As a matter of fact, this little changes never lead to systems crashing, databases being corrupted or anything like that.

But as interfaces and metadata rot, any agreements and conventions previously established will rot with them. This slowly increase the maintainability costs of the whole architecture as it creates precedents to introduce small abuses in other systems, raises the cost of integration and the need of abstraction layers between systems and, to make matters worse, making increasingly harder to prevent rot.

Alexandria documents, for example, have a metadata called "status" to control whether a document is available or not to the platform. This is not a business definition, is an architectural one. Domains are supposed to deliver only available documents if the client system doesn't ask for a specific status. One of our domains, at a given moment, implemented a state machine on top of the "status" metadata, which in turn, changed the behavior of the domain API and breaking the convention. The impact of this in all other systems was small at the time as it was a little more

painful to integrate with this Domain in particular, but eventually led to some Domains trying (and being denied) to implement business-specific states into a metadata that was supposed to control a system behavior.

This does not means that there should be no evolution to the set of interfaces and metadata of the whole platform. But this should be done considering the whole ecosystem and not to optimize one system locally. The common interfaces and metadata are the tools in place to keep the system of systems glued together and thus must be kept sane.

Documentation is a must have Because REST is simpler than the SOA model based on SOAP and the WS-* standards, we assumed in the beginning that a very thorough documentation wouldn't be needed. Developers would just get REST and, in doubt, would look for the adequate RFCs for clarification on Headers or whatever made them confused. It was a mistake.

First of all, there's a lot of misinformation in the Web about REST. This alone already leads to a significant problem of understanding and communication. Also, different people with different past experiences and professional backgrounds will have different, and some times conflicting, interpretation the specs.

Second, and most important, REST defines a style of architecture to build hypermedia-driven systems. That's all. Everything else, how documents will be represented, which (and how much) metadata will need to be added to documents, in order to make them usable to any one in the ecosystem, how will a document declare its type and why should be care about a self-descriptive document, all this things need not only to be defined and agreed upon, but be clearly communicated to all developers and professionals that work with the platform.

At last but not least, decisions must be documented, including the trade-offs, to allow any one to revisit.

What we learned is that documentation is a must have in a large-scale distributed system like Alexandria. And while it is possible to build a system this size without documentation, the amount of time wasted by not having it will quick add up in cost.

Monitoring is an unceasing effort Single applications or small-scaled systems have well defined nodes and interactions, known bottlenecks and, most of the time, developers and software architects are aware of those bottlenecks. Large-scale, distributed systems don't. This kind of system have too many moving parts, too many contact surfaces between subsystems and, as a consequence, performance hits and bottlenecks shift depending on how the whole system is being demanded.

The need to find and address performance bottlenecks in Alexandria began as soon as the first products were being delivered in production. We found out that without widespread monitoring of all systems of the platform, it would be impossible to understand what was causing a hit, whether it was a single system responsible for a perceived performance degradation or if it there were more systems to blame. It was also very hard to trace back an improvement to a set of changes or to visualize any side-effects caused by a new feature or a system update.

The bottom line is that there's no purpose in building a platform that hampers the development of new, novel digital products. But novel products will almost always

require you system to interact in unexpected ways, leading to unforeseeable effects. To be able to adjust to this effects, this flexibility comes with the price of not knowing beforehand where there can be a performance bottleneck or even a rupture to a system. The only way to deal with this, to understand what is going on and to react to any undesired behavior, is to monitor and measure continually and unceasingly.

10.7.2 Future Improvements

Besides delivering new features to support business requirements, there are two improvements in line to be implemented in the architecture of the Platform, they are:

Better HTTP Network Interface vanilla HTTP libraries found in most programming languages are limited in the way they use the network. Few implement HTTP Keep-Alive and most, if not all, do not handle parallel requests in an efficient manner. While the properties of the architecture guarantee that we will be able to scale horizontally if the need comes, as our systems gets bigger with addition of more Domains and Services, perform HTTP request more efficiently will be crucial to keep network costs low and client-server efficiency high.

Web Semantics the next step in self-descriptive messages is add semantic context to our resources. This will add an important dimension in our Platform, as it will improve the way Alexandria delivers content to both human users (journalists, editors, webmasters, end-users) and machines (services, external clients), using context-rich tools instead of plain and simple keyword matching.

The properties achieved by applying the REST constraints and the nature of the architecture gives us the flexibility to add this improvement in an incremental and reliable way.

Chapter 11
In-Process REST at the BBC

Marcel Weiher and Craig Dowie

11.1 Introduction

In the summer of 2003, the authors were hired by BBC News Interactive as team leader and technical architect to head up the so-called Sport Stats team and improve or replace the system the team was responsible for, a feeds processing platform for transforming structured XML information of live sporting events to HTML output for the BBC website in soft[1] real time.

The reasons for replacing the existing system were manifold: it was quite resource intensive, effectively unmaintainable, extremely unreliable with usually several failures per day and not capable of actually keeping up with the feeds in real time, sometimes getting backlogged by several hours or failing completely.

In addition the team itself, consisting primarily of junior level programmers, was shell shocked by working on a system where every code change would almost invariably lead to new failures, and of course the BBC Sports Interactive editorial team was also not happy with the services provided, and though there were requirements for new services, new sports and new output media such as the UK Teletext system Ceefax or Wireless Application Protocol (WAP) for pre-smartphone mobile phones, achieving those goals with the current system seemed impossible.

In the process of replacing the system, we discovered that our design principles drove us away from the very typical enterprise architecture we had originally envisaged towards an architecture that appeared very REST-like, despite not being

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

[1] Results degrade significantly in value if they are late, but lateness does not constitute total system failure (hard real-time) or reduce the value of results to zero (firm real-time).

M. Weiher (✉)
Metaobject Ltd., 26 York Street, London W1U 6PZ, United Kingdom
e-mail: marcel@metaobject.com

C. Dowie
Betfair, The Waterfront Winslow Road London, London W6, United Kingdom
e-mail: craig.dowie@betfair.com

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_11,

distributed. As we embraced some of these REST-like qualities, our system became simpler, faster and more robust.

The remainder of this chapter will describe the task to be solved, describe the original system and analyze its shortcomings (Sect. 11.2). Then we describe the overall architecture (Sect. 11.3) and implementation (Sect. 11.4) of our replacement system. After presenting the results we got in Sect. 11.5 and performing a more in-depth evaluation in Sect. 11.6. we conclude with a look at related work and a look at possible implications of our work.

11.2 The Task and Its First Solution

The BBC uses feeds from different sports data providers to deliver basic sports information such as schedules, scores, tables and live updates during matches. This information is provided using XML files, which are provided via FTP, ingested into the system and used to update an internal model of all the sporting events and their associated metadata, such as teams, players, leagues and competitions.

Arrival of data not only causes updates to the model, but also trigger output of the pages affected by the incoming data. Pages are written to a dedicated export system that is mirrored to the live-servers that serve static HTML pages.

11.2.1 Version 1

The existing system that was to be replaced was SportStats version 1, the replacement project was SportStats version 2, or just v2 as the team tended to call it. Version 1 consisted of more than 100 processes, instances of 8 different types of programs, running on 10 Windows machines, all backed by an Oracle SQL database server running on a Sun SPARC server.

The central components of the system were implemented in around 40 thousand lines of Objective-C code using version 4.5 of Apple's *WebObjects*[2] web application server, Enterprise Objects Framework Object Relational Mapper (*ORM*) and several helper programs written in Perl, mostly for fetching files from FTP servers and moving them to different feed directories.

A number of *Populator* processes would watch the feed directories, parse any XML files found there, populate the database with the information and queue up trigger information for *Builder* processes. The *Builder* processes would poll the database for these triggers, and then use HTTP requests to *Renderer* processes to actually generate the HTML, capture their output and then place the generated files on an output file system, along with metadata that would tell the output processes (part of the BBC infrastructure and not part of SportStat) where to place the HTML files on the live site.

Figure 11.1 is part of the original documentation of the v1 system and is included to illustrate the complexity of the system as deployed. It shows the processes involved in generating a small part of the football site and only includes the *Populator* and *Builder* processes for that part of the site, *Renderers* are not included.

[2] http://www.apple.com/ca/webobjects/

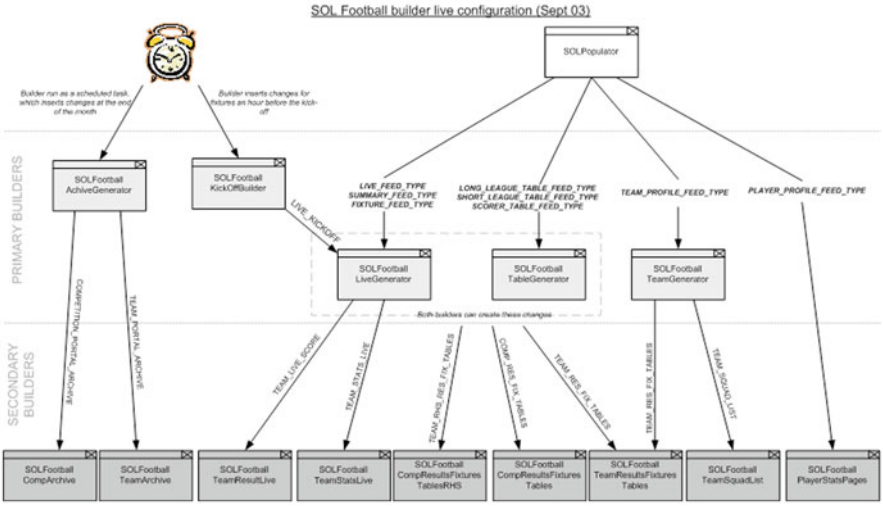


Fig. 11.1 Original diagram describing part of the deployment architecture of v1

The *Renderer* processes had MVC-style templates for each different kind of HTML page produced. The URLs used to interact with the *Renderer* processes also reflected the MVC style, effectively encoding the template kind, not the location of the document in the output.

Due to the fact that WebObjects is an web *application* framework, actually producing HTML output was considered part of the view, and owned by the framework's request/response loop. This made republishing difficult in case there was an inconsistency on the site: the only way to republish a page was to rerun the feed input and hope that it would trigger the output again.

As mentioned earlier, rendering was quite slow, taking anywhere from two seconds to two minutes to generate a single page, and that's not counting the time taken by the FTP processes, *Populators* and *Builders*. Adding more machines and processes didn't seem to improve the situation.

The whole system had no automated tests.

11.2.2 Analysis

A couple of things stood out immediately. First, there simply is no good reason why generating around 100 kb of HTML should take two minutes, or over a millisecond per byte on 1 GHz class machines.

Second, there were way too many boxes and processes, and these were not helping performance, but hurting it. Another problems was the overuse of the relational database, not just for sports data, but also to act as a message queue. One of the reasons adding machines never helped was that the database server was saturated at 100 % CPU and I/O load.

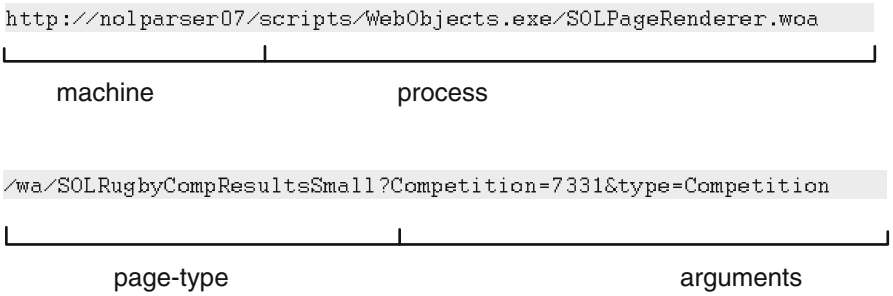


Fig. 11.2 URLs used in SportStats v1

While use of SQL databases was pretty much standard in enterprise settings, we were getting very little actual benefit from the database: since we were not the originators of the data, but just republishers, we couldn't really maintain any sort of meaningful integrity even if we wanted to. Additionally, sports data is highly irregular, so relational modeling proved impractical for most data, which was instead maintained as serialized dictionaries that were stored as BLOBs in the database.

While the use of URLs by *Builders* to request specific HTML from their *Renderers* seemed promising in terms of RESTful implementation, in practice it combined the worst of both worlds: the URLs (see Fig. 11.2) were dictated by the application framework and action-oriented rather than resource-centric, specifying the process and specific page-type to render rather than the document's location, which was specified elsewhere. On the other hand, the way the application framework had encapsulated URL handling and the request/response loop meant that generating HTML pretty much required the two separate *Builder* and *Renderer* processes mentioned above.

What we wanted was the exact opposite: resource-oriented URLs, but only a single process, rather than action-oriented URLs with multiple processes.

11.2.3 Solution Sketch

Since performance was a major goal of the rewrite and a message-send inside a process is around 1 million times faster than a network hop (nanoseconds vs. milliseconds), one of the main aims of the rewrite was to keep as much processing inside a single process as possible. Another impetus for this was the desire to construct the system using a Test Driven Development (*TDD* [21]) style, which also strongly favors having all functionality unit-testable, rather than having system-level or integration tests.

Although we wanted to keep using the web application framework in question for example in order to make use of the templating system that was well known within the organization (though updated to the 5.0 version based on Java instead of Objective-C), we wanted to have URLs that reflected the final output's document structure rather than program structure.

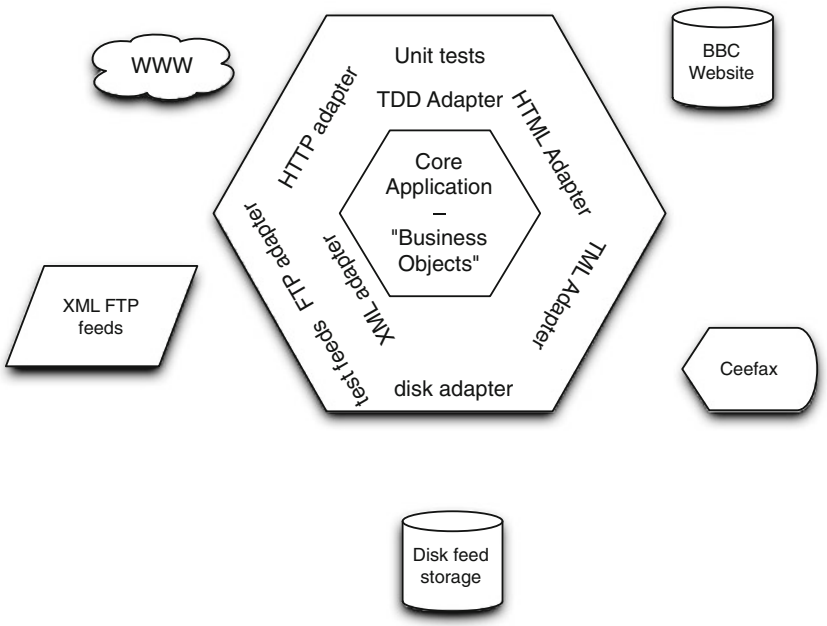


Fig. 11.3 Hexagonal Architecture

Due to the irregular structure of sports data mentioned above, modeling was to be object-oriented instead of relational. The SQL database would be used as a key-value store with minimal structuring, and we would keep all the original feed data in order to have full accountability and traceability.

Although we expected the architecture to help with our performance goals, we expected most of the gains to come from improved code quality, which experiments with the v1 system and selected implementation spikes for the new architecture confirmed as feasible.

11.3 Architecture

Our desire for performance and testability drove us towards what is now known as *Hexagonal Architecture*[3], also known as *Ports and Adapters*: all relevant functionality is provided as callable and testable APIs in the system's programming language, so in our case Java. All interfaces to the outside world, be they input, output or data storage, are provided via thin and semantic-free wrappers. Figure 11.3 shows the architecture adapted to the SportStats v2 system.

[3] http://alistair.cockburn.us/Hexagonal+architecture

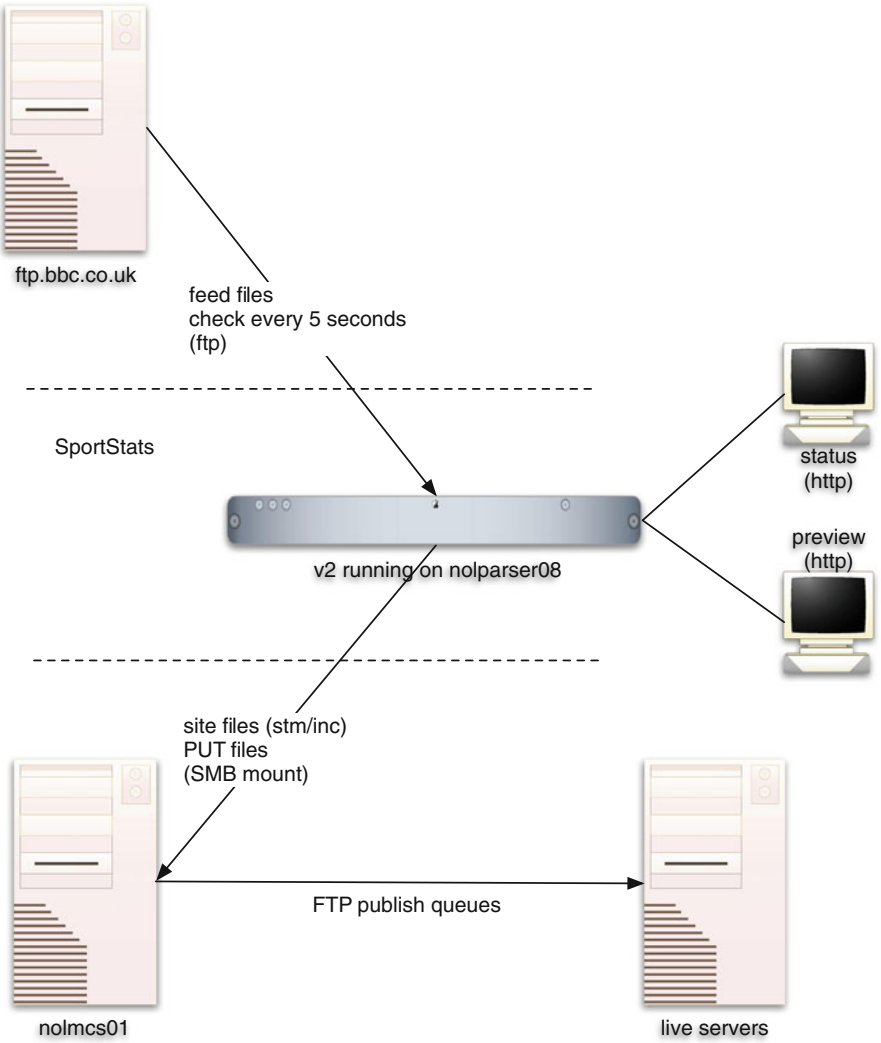


Fig. 11.4 Deployment architecture of SportStats v2

By keeping functional units in-process, the deployment architecture of SportStats v2 could be dramatically simplified. Figure 11.4 is part of the original v2 documentation showing the complete deployment architecture of SportStats v2: a single machine running a single Java process processing both the input feeds and providing monitoring. Compare this to Fig. 11.1, which shows just a small part of the v1 system, whereas the v2 diagram was actually expanded a bit to show machines responsible for monitoring and other infrastructure in order to have more than a single box.

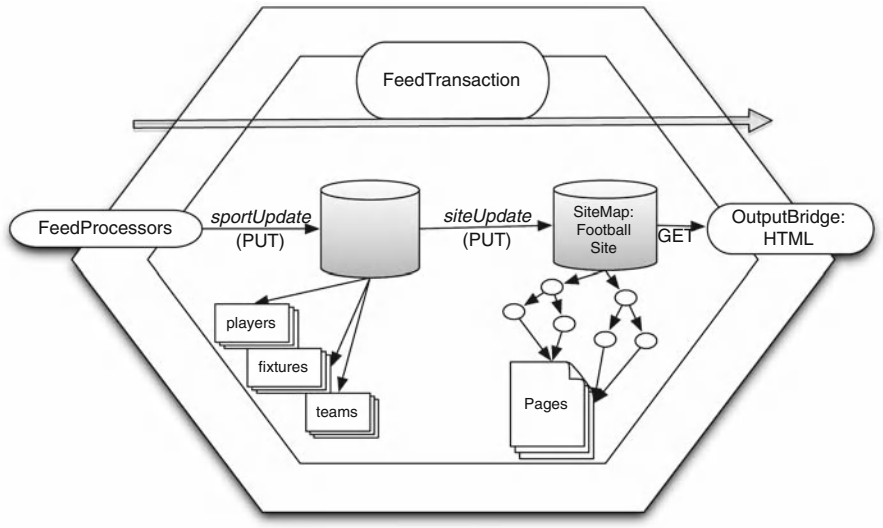


Fig. 11.5 Internal architecture of SportStats v2

In keeping with the XP best practice of only implementing what is absolutely necessary, we discovered that after we had implemented the feed reader (necessary) and the internal model (necessary), we actually had a working system without ever getting around to hooking up the database, which we promptly removed from the architecture.

Since we were keeping a safety log of the incoming feeds and processing those feeds was very quick, we could recover our initial state by simply replaying those feeds to the system with output disabled.

This architecture, with an in-memory model and an external source of events that affect the model and can be replayed was later given the name *EventPoster*[4] by Martin Fowler, so what we really have is a combination of *EventPoster* and *Hexagonal* architectures, two styles that fortunately aren't inconsistent with each other.

Keeping to the theme of simplicity, we wanted our internal structure to map as directly to our result as possible. Figure 11.5 shows the result using the example of a Football site: the core application consists of a *Sport* object and a *SiteMap*. The former models all the background information about a particular sport, the latter the actual output that is supposed to exist on the live servers.

Following the hexagonal architecture and our desire for simplicity, the objects that the *Sport* maintains are essentially 1:1 representations of the XML feeds received, whereas the *SiteMap* is a 1:1 representation of the particular website. All semantically relevant transformations occur between the *Sport* and the *SiteMap*, without any reference to external representations.

[4] http://martinfowler.com/bliki/EventPoster.html

The *FeedLoop* (not shown) drives individual *FeedTransactions* through the system, from the input ports through the core system to the output ports. URIs are the lingua franca used throughout the system: they define which files need to be fetched, which nodes of the *SiteMap* need to be updated, for which nodes new output needs to be generated and where that output needs to be written. The *FeedTransaction* keeps track of these changing sets of URIs as processing ripples through the system.

It should be noted that the externally imposed interfaces of the system are not those of a traditional RESTful server: rather than waiting for either PUT/POST or GET requests, it both actively pulls source data via FTP and actively pushes results, also via FTP.

Had we followed the REST style in a more traditional fashion, both the *Sport* object and the *SiteMap* object would have been modeled as REST servers running in separate processes, with the *FeedLoop* an external process that manages the URIs and drives the other components via HTTP. However, this seemed like complexity that was both wholly unnecessary and also conflicted with our goals of unit-testability and performance, so we instead moved those APIs in-process.

Figure 11.6 shows the evolution of the architecture from the process- and database-centric structure in V1 via a more traditional multi-process REST-like architecture to the single process that we ended up with.

The initial move to a more RESTful architecture sees a reduction in the number of processes and a move away from active processes and passive data-stores to servers that actively model/manage specific types of data and provide document-centric interfaces for them.

Many of the architectural benefits we saw came from switching to a RESTful architecture, but there didn't seem any good reason for requiring multiple processes just to adhere to the idea that RESTful interfaces must match up to process boundaries, so our final step was to simply move those interface into the process.

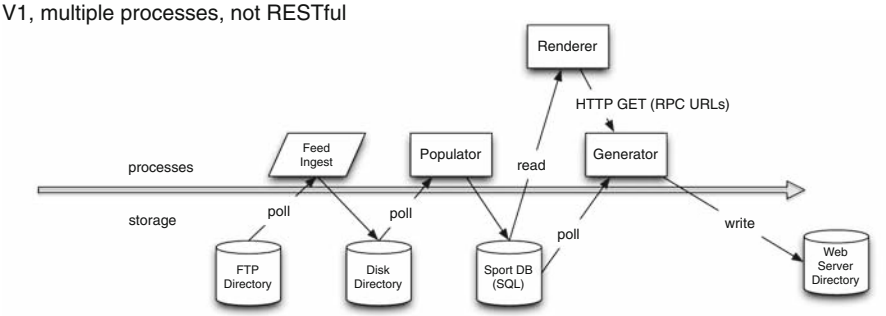
Figure 11.7 shows two *FeedTransactions* as displayed by the system's HTML logging console, which simply runs as another port into the core system. For each FeedTransaction it shows what input files were processed and what output pages were affected, in addition to various timings that give an indication of whether the system is healthy or not. All the *affected pages* listed are clickable hyperlinks that are rendered live by the system from its internal state.

11.4 Implementation

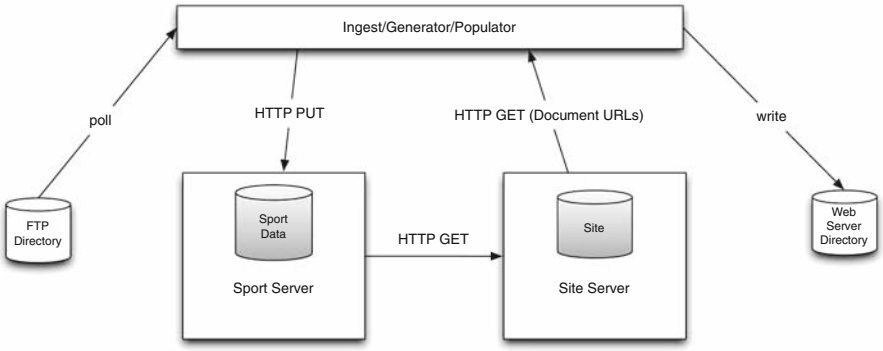
The architecture just described was implemented as a single Java application using the WebObjects 5 web framework with a total of around 10 thousand lines of production code and 2000 lines of testing code in more than 200 individual tests. More precisely, the architecture was discovered incrementally as we implemented the system following our guidelines and principles as well as the business requirements.

There is no database, no RPC or distributed processing of any kind and no multi-threading of the processing loop. In-memory performance is so much faster than I/O that multi-threading was simply not necessary. Deployment is a single java archive (*JAR*) along with a platform specific script to run the JAR.

V1, multiple processes, not RESTful



V2 RESTful, multiple processes



V2, RESTful interface in-process

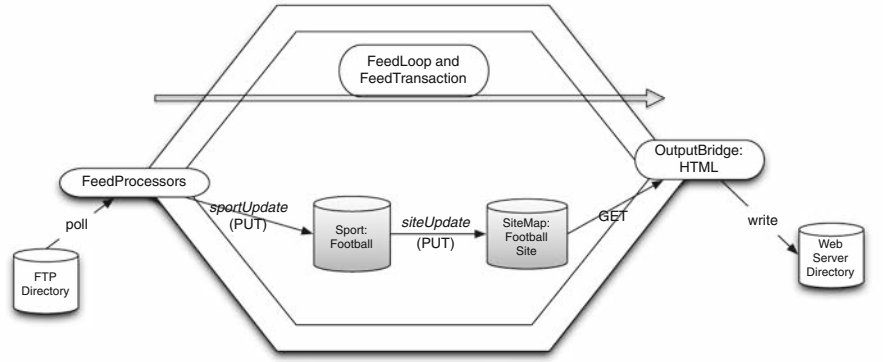
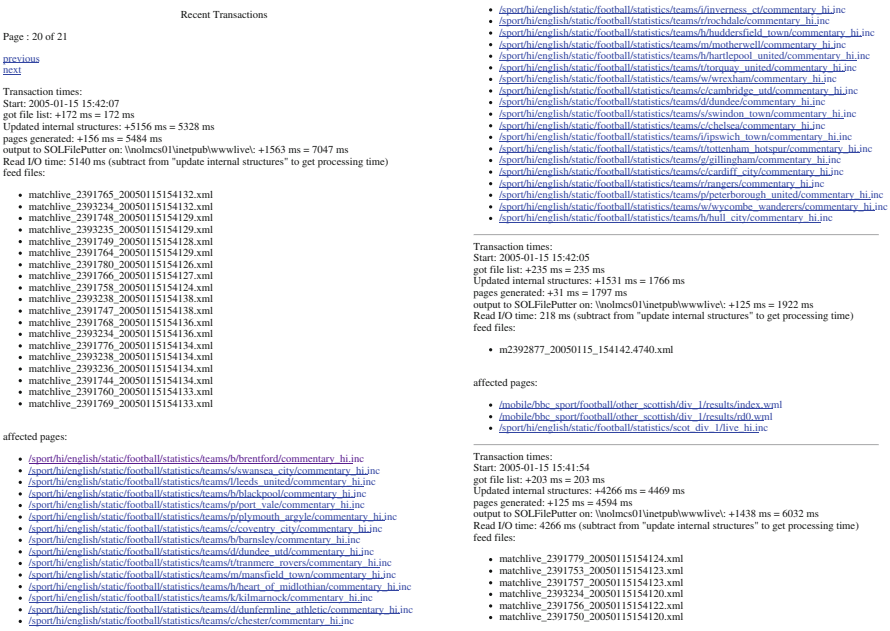


Fig. 11.6 Architecture evolution from processes to In-Process REST



Recent Transactions

Page : 20 of 21

previous
next

Transaction times:
Start: 2005-01-15 15:42:07
got file list: +172 ms = 172 ms
Updated internal structures: +5156 ms = 5328 ms
pages generated: +156 ms = 5484 ms
output to SOLFilePutter on: \\nolmcs01\inetpub\wwwlive\: +1563 ms = 7047 ms
Read I/O time: 5140 ms (subtract from "update internal structures" to get processing time)
feed files:

- matchlive_2391765_20050115154132.xml
- matchlive_2393234_20050115154132.xml
- matchlive_2391748_20050115154129.xml
- matchlive_2393235_20050115154129.xml
- matchlive_2391749_20050115154128.xml
- matchlive_2391764_20050115154129.xml
- matchlive_2391780_20050115154126.xml
- matchlive_2391766_20050115154127.xml
- matchlive_2391758_20050115154124.xml
- matchlive_2393238_20050115154138.xml
- matchlive_2391747_20050115154138.xml
- matchlive_2391768_20050115154136.xml
- matchlive_2393234_20050115154136.xml
- matchlive_2391776_20050115154134.xml
- matchlive_2393238_20050115154134.xml
- matchlive_2393236_20050115154134.xml
- matchlive_2391744_20050115154134.xml
- matchlive_2391760_20050115154133.xml
- matchlive_2391769_20050115154133.xml

affected pages:

- /sport/hi/english/static/football/statistics/teams/b/brentford/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/s/swansea_city/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/l/leeds_united/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/b/blackpool/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/p/port_vale/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/p/plymouth_argyle/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/c/coventry_city/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/b/barnsley/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/d/dundee_utd/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/t/tranmere_rovers/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/m/mansfield_town/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/h/heart_of_midlothian/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/k/kilmarnock/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/d/dunfermline_athletic/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/c/chester/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/i/inverness_ct/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/r/rochdale/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/h/huddersfield_town/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/m/motherwell/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/h/hartlepool_united/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/t/torquay_united/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/w/wrexham/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/c/cambridge_utd/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/d/dundee/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/s/swindon_town/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/c/chelsea/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/i/ipswich_town/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/t/tottenham_hotspur/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/g/gillingham/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/c/cardiff_city/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/r/rangers/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/p/peterborough_united/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/w/wycombe_wanderers/commentary_hi.inc
- /sport/hi/english/static/football/statistics/teams/h/hull_city/commentary_hi.inc

Transaction times:
Start: 2005-01-15 15:42:05
got file list: +235 ms = 235 ms
Updated internal structures: +1531 ms = 1766 ms
pages generated: +31 ms = 1797 ms
output to SOLFilePutter on: \\nolmcs01\inetpub\wwwlive\: +125 ms = 1922 ms
Read I/O time: 218 ms (subtract from "update internal structures" to get processing time)
feed files:

- m2392877_20050115_154142.4740.xml

affected pages:

- /mobile/bbc_sport/football/other_scottish/div_1/results/index.wml
- /mobile/bbc_sport/football/other_scottish/div_1/results/rd0.wml
- /sport/hi/english/static/football/statistics/scot_div_1/live_hi.inc

Transaction times:
Start: 2005-01-15 15:41:54
got file list: +203 ms = 203 ms
Updated internal structures: +4266 ms = 4469 ms
pages generated: +125 ms = 4594 ms
output to SOLFilePutter on: \\nolmcs01\inetpub\wwwlive\: +1438 ms = 6032 ms
Read I/O time: 4266 ms (subtract from "update internal structures" to get processing time)
feed files:

- matchlive_2391779_20050115154124.xml
- matchlive_2391753_20050115154123.xml
- matchlive_2391757_20050115154123.xml
- matchlive_2393234_20050115154120.xml
- matchlive_2391756_20050115154122.xml
- matchlive_2391750_20050115154120.xml

Fig. 11.7 FeedTransaction log

The architecture diagram in Fig. 11.5 is organized by the FeedTransaction that moves data through the system. In what follows, we will use the same organization to explain the implementation in more detail.

11.4.1 Feeds and Domain Objects

Incoming XML feeds are converted to mostly data-bearing domain objects such as a Player, a Team, and a Fixture by various individual *FeedProcessors*. After generating the domain objects, the processors pass them on to their corresponding *Sport* objects via the SportUpdate interface.

Specific Sport objects such as Football or Cricket maintain all that is known about a particular sport, in essence they store all the most current versions of the incoming feed data.

11.4.2 SiteMaps

Each of several *SiteMap* object holds a tree of *PresentationObject* objects that represent the actual pages the system should generate for a particular site. Each

PresentationObject object is connected to its parent, recursively up to the root object of the tree, and can potentially have named child objects, just like a filesystem. The *SiteMap* objects have methods finding *PresentationObjects* by URI (public PresentationObject presentationObjectForUri(String path)), but tree management itself is outsourced to the *PresentationObjects* themselves.

A *MultiSite* allows different sites to be combined into a single composite site, similar to the Unix mount command.

The different *Sport* objects maintain state in their respective *SiteMaps* via the SportUpdate protocol, which usually updates single *PresentationObjects* with a new domain object, essentially like an HTTP PUT. Each modified object notes its paths as having been modified in the FeedTransaction.

Effectively, the *SiteMap* is a complete, fully expressed model of the web-site we wish to maintain, but stored in memory. This in-memory web-site gets set up once at startup and afterwards gets updated via the equivalent of PUT requests, and when updates are complete, gets mirrored to the actual web-site using the bridges.

11.4.3 Bridges and Templates

Once all the updates to the internal model have been performed, the FeedTransaction proceeds to generate the output for all the *PresentationObjects* that have changed. To do this, it uses the list of changed URIs to effectively perform a GET against the respective *SiteMap*, but this time via an *OutputBridge*. The OutputBridge in turns uses that URI to fetch the *PresentationObject* in question and then does bridge-specific processing to generate an output-representation, in the case of HTML it applies an HTML template.

This corresponds closely to the role of the *Builder* processes in SportStats v1, and in fact even the processing performed is very similar: an HTML template with variables is applied to *PresentationObject* and the result returned. However, the semantics are quite different, in v1 a specific type of page-renderer was asked to create a type of HTML page, whereas in v2 we are simply retrieving a resource, a page of our site that we want in a specific representation. In addition, v2 does all of this processing in memory, whereas in v1 the *Builder* process had to send the URI to a *Renderer* process.

The need to run in memory was driven primarily by the requirement of having a completely unit-testable system. Having to run in memory required major surgery to parts of the web framework in order to decouple the rendering from the request handling, because we needed to render HTML templates without having an external request. Having removed the request handling, we no longer had framework-defined, action-oriented URLs to deal with, but instead were faced with the task of having to define what those URLs should look like ourselves, and in this case, why not make them document-oriented instead?

Once we were running in memory, we noticed that performance was simply no longer an issue, so we could simplify the system down to a single process, running only a single processing thread.

11.4.4 Ceefax and WAP

Ceefax, the BBC's videotext system which ran from 1974 to 2012, was straightforward to support, despite having a completely different structure. Ceefax, which is transmitted in the vertical blanking interval of analog television sets, is organized into numbered pages, each of which has a fixed 40 character by 24 row format. The individual pages are defined in a binary-coded format similar to terminal control sequences, so nothing like HTML.

After defining a simple one-level URI namespace for Ceefax pages and a special output mapper that translated our *Videotext Markup Language* into command sequences for the *Plasma* system that maintained Ceefax, working with Ceefax was just like working with the website: the *Sport* would update the *SiteMap* representing Ceefax pages, and those pages would then get mirrored to the Plasma system.

The *Wireless Application Protocol* (WAP) was even more straightforward to support than Ceefax, due to the fact that the *Wireless Markup Language* is just an XML dialect and could therefore be supported using our existing templating logic.

11.5 Results

SportStats v2 was highly successful in meeting its goals, improving on the original system in all key metrics, sometimes significantly, as shown in Fig. 11.8.

Performance was improved an average of 200-fold, so the system was no longer the bottleneck and instead started putting a strain on other parts of the BBC infrastructure that so far had been shielded from the onslaught of real time sports data by the weakness of the old system. For example, we started hitting the ephemeral port limitations of the standard FTP protocol that was used both on the feed input and page output side.

It should be noted that the application code itself received hardly any optimization work, at times being almost willfully non-optimized. The performance gains we experienced were due almost exclusively to the architecture in question, allowing day-to-day work to almost completely ignore performance concerns.

Major failures, which had previously been an almost weekly occurrence disappeared almost completely. The three failures that did happen over the course of two years were the following:

1. A Java VM configuration error. The team's experience with Java deployment was limited so no one knew that the default maximum heap size for a VM was only 64 MB. Since we had an entire machine to ourselves we subsequently set the heap size to 2 GB and never had a problem again.
2. UI code that hadn't been unit tested and had a significant bug, made worse by having untrained users interact with the buggy UI. We learned more about factoring out the functionally significant parts of UI code into unit-testable parts.

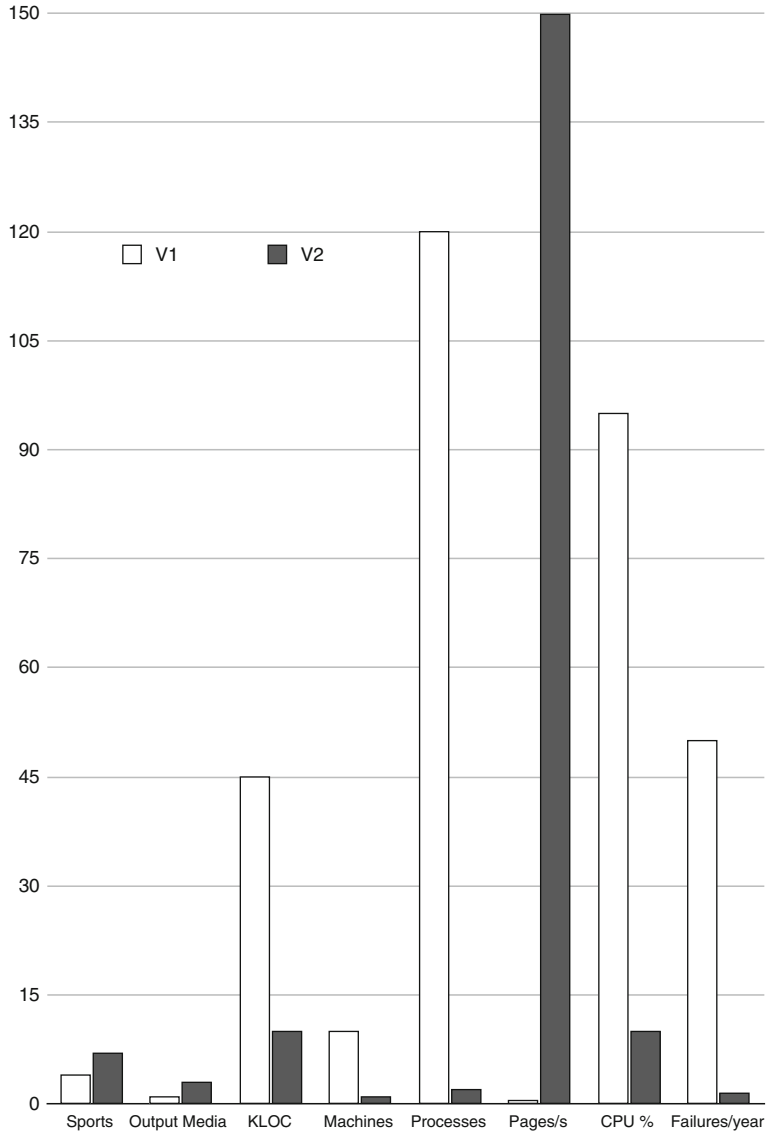


Fig. 11.8 Comparison of V2 with V1

3. A hiccup in our incoming FTP chain caused feed files to back up. Once the files had piled up, our I/O bandwidth was insufficient to catch up. We implemented an addition to our FTP processor that identified obsolete data in the FTP directory by filename alone and deleted old files if newer data was also available at the same time. The same technique was also used to prune the feed archive.

Nice side effects included that both code bulk and resource usage were reduced drastically, from 40 KLOC to 10KLOC and from 10 machines to only one required for the system, though we eventually deployed on two machines in order to have a failover option. Relieving the team from emergency response and repair work allowed much more editorial work to occur, adding new sports, competitions or output media, but more importantly it also boosted team morale, confidence and productivity.

11.6 Evaluation

Although we did not set out to apply REST principles in-process, and in fact most of the team hadn't even heard or never mind read Fielding's thesis [76], our other architectural goals and principles slowly pushed us into that direction: on the one hand, a REST-like approach promised to greatly simplify our system, on the other hand staying in-process greatly increased testability and performance and further reduced complexity.

The idea of organizing mutable application state as resources that can be navigated using paths and queried/updated using clearly defined operations like GET, PUT and POST, with similar semantic guarantees seems very powerful to us. Having such a store, where objects have *place* in addition to *identity* has obvious organizational benefits, and building a system in this way shouldn't require splitting it into multiple processes, just so communication can be HTTP between processes. Having the in-memory representation be the primary one generally makes *domain modeling* [71] easier, and certainly did in our case.

11.6.1 REST Constraints

The REST architectural style consists of a number of constraints. Here, we discuss to what extent SportStats v2 did or did not adhere to those principles and what the impact was.

Client-server

This is the constraint we purposely did not adhere to, at least in the traditional sense of client and server being separate computers or at least processes: our *FeedTransaction* object is the client, the *SiteMap* and *Sport* objects the servers, and they all co-exist in the same process.
However, having objects stand in for actual computers is one of the quintessential features of object-oriented programming: "Smalltalk semantics are a bit like having thousands and thousands of computer all hooked together by a very fast network" [117]. By this definition, we have adhered to the client-server constraint, even if we do not have separate client and server computers.
Determining whether either definition of client-server

Stateless	The state of the transaction is fully contained in the *FeedTransaction* object, the servers do not maintain any of this state though they are modified in response to it.
Cacheable	The fact that the servers are stateless makes them cacheable. In fact, an experimental version of the system that directly served HTML to clients rather than pushing it to distribution servers added a cache for the generated HTML, thus pushing performance to the tens-of-thousands requests per second.
Layered system	Our internal *SiteMap* were actually later combined into *MultiSite* objects that distributed incoming requests to their sub-sites, with clients being no wiser as to whether they were dealing with a *MultiSite* or a *SiteMap*.
Code on demand (optional)	This optional constraint was not applicable to our system.
Uniform interface	Using URIs uniformly throughout the system was one of the driving factors of the design and had the most visible benefits, such as introspection, ease of construction and maintenance, logging and debugging. However, we did not apply this principle as uniformly as we would have liked. For example, the connections between the *Sport* objects and both the feeds themselves and the *SiteMap* were standard Java interfaces rather than actual PUTs with URIs and update objects. Hypermedia as the Engine of Application State (HATEOAS) was at least partially adhered to, as the *FeedTransaction*'s state consisted mostly of URIs that indicated what pages to fetch and where to deliver them, and those URIs were provided by the servers. These URIs did not contain the next action to take, that was hard-coded in the *FeedTransaction*, but the actions were really trivial: push update, fetch changed URIs, fetch results from URIs.

Overall, we were able to adhere fairly closely to the REST constraints as outlined, and adhering to those principles made our system better despite not being a classic HTTP-based client server system. The question becomes whether *client-server* needs to be defined in the classic networking sense for REST-conformance or can be loosened to the object-oriented definition of *client-server*.

This adherence to the REST constraints mostly applies to the in-process interfaces we have been discussing. Our external interfaces were pre-defined legacy and not RESTful at all, with the system actively pulling and pushing data via FTP, rather than being a server responding to GET and PUT/POST requests. Although the downstream interface could easily have been converted to use HTTP GET, and in fact was for monitoring and as an experiment, the upstream interface actually benefitted from active polling: combining multiple new feed files into a single transaction smoothed

out both load spikes and service interruptions, and allowed our downstream infrastructure to throttle the load produced by our system. Reacting to every individual feed file sent via a POST would have drastically increased our downstream load, possibly beyond the capacity of the available infrastructure.

11.6.2 Drawbacks

One of the drawbacks of this approach was that the URIs that essentially become the identifiers used in our system were mostly strings, which had to be processed at runtime and passed as arguments to methods dealing with them. Having language support for user-defined, rich identifiers in URI form would have been very helpful.

Another disadvantage of our approach is that the application state must actually fit in memory. Apart from configuration errors, this turned out not to be a problem for us, and with multi-gigabyte main memories now standard equipment, even commodity hardware is able to accommodate a large subset of applications. Restoring application state from secondary storage can be more problematic: our system sometimes required over a minute to re-read its archived feed, but this was sufficiently fast for our requirements that we only performed very rudimentary optimizations.

11.7 Related Work

The lack of scalability of database- and I/O-centric architectures has recently led to developments such as the memcached [5], caching layer, which is often seen as crucial to the scalability of popular website, but also requires modification to actually handle the load[6]. In fact, it is so crucial that in the end, the caching layer often becomes the primary storage mechanism, at which point the complexity of the tiered system becomes increasingly questionable.

Project Voldemort is one attempt at reducing this complexity by designing for in-memory operations from the start.[7] Storage is pluggable and needs not be disk-based. VoltDB[8] similarly claims massive reductions in complexity and increases in performance by designing a relational database from the start for (single-threaded) in-memory operations. A project taking this type of approach much further than we did is the LMAX trading system created by Betfair and described by Martin Fowler[9]. They claim 6 million transactions per second throughput on commodity hardware.

However, none of these in-memory systems appear to have adopted a REST-like internal storage model.

---

[5] http://memcached.org

[6] http://www.facebook.com/note.php?note_id=39391378919 and http://engineering.twitter.com/2012/07/caching-with-twemcache.html

[7] http://www.project-voldemort.com/voldemort/

[8] http://www.voltdb.com/

[9] http://martinfowler.com/articles/lmax.html

Some of the new NoSQL databases such as CouchDB[10] do sport document-based interfaces, but do so only via a HTTP, so are not used in-process.

The Plan 9 operating system [190, 191] tried to organize as many resources and services in hierarchical file-systems as possible and had a per-process unified file-system that acted as a name-space for organizing these resources, but those resources were still managed by servers outside the process, with access mediated via the kernel.

11.8 Conclusion and Outlook

Working on our (replacement) feeds processing application, we found that design principles such as testability and simplicity drove us towards both a REST-like design and a fully in-process model, and consequently to applying REST-like principles to a non-distributed system. The approach proved to be surprisingly fruitful in terms of simplicity/tractability, performance, reliability and productivity.

Looking back, it probably would have been helpful to apply the principles even more broadly, for example also organizing the domain objects stored in the different *Sport* objects as a tree with path access, rather than as nested dictionaries. This would likely have reduced the number iterations we had with this design and made the system even more uniform, but was difficult due to the lack of language support for URI-based interfaces.

Better language support for URIs to be expressed directly rather than specified indirectly through strings and helper path objects would also have helped, because the very clean and simple path-based access structures were often hidden in the string-manipulation code required to create or access them.

Overall, we were very encouraged by how well REST principles scaled down to modeling parts of a program or system that are not distributed, in addition to their well-documented success for distributed systems. We will be using this approach in future system and look forward to better library and language support, as well as more research into consequences and theoretical foundations.

[10] http://couchdb.apache.org

References

1. Paul Adamczyk, Munawar Hafiz, and Ralph Johnson. Non-compliant and Proud: A Case Study of HTTP Compliance. DCS-R-2935, University of Illinois, August 2008. http://hdl.handle.net/2142/11424. 53
2. Ben Adida, Mark Birbeck, Shane McCarron, and Iván Herman. RDFa Core 1.1—Second Edition: Syntax and processing rules for embedding RDF through attributes. World Wide Web Consortium, Proposed Edited Recommendation PER-rdfa-core-20130625, June 2013. 60, 61
3. Gul Abdulnabi Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press Cambridge, MA, USA, 1986. 14
4. Ashish Agrawal, Mike Amend, Manoj Das, Mark Ford, Chris Keller, Matthias Kloppmann, Dieter König, Frank Leymann, Ralf Müller, Gerhard Pfau, Karsten Plösser, Ravi Rangaswamy, Alan Rickayzen, Michael Rowley, Patrick Schmidt, Ivana Trickovic, Alex Yiu, Matthias Zeller. WS-BPEL Extension for People (BPEL4People), Version 1.0. Retrieved on November 2012 from http://docs.oasis-open.org/bpel4people/bpel4people-1.1.html, 2007. 135
5. Rosa Alarcón, Cesare Pautasso, and Erik Wilde, editors. *Third International Workshop on RESTful Design (WS-REST 2012)*, Lyon, France, April 2012. xvii, 199
6. Rosa Alarcón, Cesare Pautasso, and Erik Wilde, editors. *Fourth International Workshop on RESTful Design (WS-REST 2013)*, Rio de Janeiro, Brazil, April 2013. xvii
7. Omar Aldawud, Tzilla Elrad, and Atef Bader. UML Profile for Aspect-Oriented Software Development. In *The Third International Workshop on Aspect Oriented Modeling*, 2003. 115
8. Christopher Alexander. *A Timeless Way of Building*. Oxford University Press, 1979. 82
9. Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2004. 123
10. Scott W. Ambler. *The Elements of UML 2.0 Style*. Cambridge University Press, 2005. 120
11. Scott W. Ambler. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press, 2012. 122, 124
12. Mike Amundsen. *Building Hypermedia APIs with HTML5 and Node*. O'Reilly Media, Inc., November 2011. 49, 88
13. Mike Amundsen. Collection + JSON—Hypermedia Type. Retrieved October 29, 2012, from http://amundsen.com/media-types/collection/, 2011. 95
14. Mike Amundsen. Hypermedia Types. In Wilde and Pautasso [250], pages 93–116. 88
15. AndroMDA. What is AndroMDA?, 2012. 127. http://www.andromda.org/
16. Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 2011 ACM Conference on Programming Language Design and Implementation (PLDI)*, volume 46:6, pages 355–366. ACM, 2011. 16

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3, © Springer Science+Business Media New York 2014

17. Apple. *Local and Push Notification Programming Guide*. iOS Developer Library, August 2011. 52
18. Joe Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007. 16
19. Julian Aubourg, Jungkee Song, and Hallvord R. M. Steen. XMLHttpRequest. World Wide Web Consortium, Working Draft WD-XMLHttpRequest-20121206, December 2012. 73
20. Thomas Baker, Natasha Noy, Ralph Swick, and Ivan Herman. Semantic WebCase Studies and Use Cases, 2007–2012. 67
21. Kent Beck. *Test-Driven Development by Example*. Addison-Wesley, Boston, 2003. 176
22. Kent Beck and Ward Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical Report CR-87-43, Tektronix, Inc., 1987. http://c2.com/doc/oopsla87.html. 82
23. Tim Berners-Lee. Linked Data, Design Issues for the World Wide Web. Retrieved June 6, 2010, from http://www.w3.org/DesignIssues/LinkedData.html, 2006. 96
24. Tim Berners-Lee. Read-Write Linked Data, August 2009. 64
25. Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. W3C Team Submission, March 2011. 65
26. Tim Berners-Lee, Roy Thomas Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. Internet RFC 3986, January 2005. 42, 164
27. Tim Berners-Lee, James A. Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001. 65
28. Kamal Bhattacharya, Cagdas Gerede, Richard Hull, Rong Liu, and Jianwen Su. Towards formal analysis of artifact-centric business process models. In *Proceedings of the 5th international conference on Business process management*, BPM'07, pages 288–304, Berlin, Heidelberg, 2007. Springer-Verlag. 135
29. Dennis Pfisterer, Kay Römer, Daniel Bimschas, Oliver Kleine, Richard Mietz, Cuong Truong, Henning Hasemann, Alexander Kröller, Max Pagel, Manfred Hauswirth, Marcel Karnstedt, Myriam Leggieri, Alexandre Passant, and Ray Richardson, SPITFIRE: toward a semantic web of things, IEEE Communications Magazine, Volume:49, Issue: 11, Pages: 40-48
30. Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS'11)*, volume 5, pages 1–15. ACM, 2011. 17
31. Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004. 98
32. Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984. 5
33. Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data—The Story So Far. *International Journal On Semantic Web and Information Systems*, 5(3):1–22, 2009. 62
34. Allen C Bomberger, William S Frantz, Ann C Hardy, Norman Hardy, Charles R Landau, and Jonathan S Shapiro. The KeyKOS Nanokernel Architecture. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 95–112, 1992. 6
35. L. Bougé and N. Francez. A compositional approach to superimposition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'88, pages 240–249, New York, NY, USA, 1988. ACM. 136
36. Peter Braun and Wilhelm R Rossak. *Mobile agents: Basic concepts, mobility models, and the tracy toolkit*. Morgan Kaufmann, 2005.
37. Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.98. Retrieved January 17, 2011, from http://xmlns.com/foaf/spec/20100809.html, 2010. 97
38. Frederick P Brooks Jr. The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering. Pearson Education, 1995. 150
39. Paul C. Bryan and Mark Nottingham. JavaScript Object Notation (JSON) Patch. Internet RFC 6902, April 2013. 56

40. Paul Buchheit. Simple Update Protocol, 2008. 40
41. Thomas Burkhart and Peter Loos. Flexible Business Processes—Evaluation of Current Approaches. In *Proceedings of Multikonferenz Wirtschaftsinformatik 2010*, 2010. 135
42. Joseph Campbell. *The hero with a Thousand Faces*. Pantheon Books, 1949. 83
43. Jorge Cardoso. *Semantic Web Services: Theory, Tools and Applications*. IGI Publishing, Hershey, PA, USA, 2007. 135
44. Steinar Carlsen, John Krogstie, and Odd Ivar Lindland. Evaluating Flexible Workflow Systems. In Proceedings of the 30th Hawaii International Conference on System Sciences: Information Systems Track-Collaboration Systems and Technology—Volume 2, HICSS '97, pages 230–, Washington, DC, USA, 1997. IEEE Computer Society. 135
45. Fabio Casati, Ski Ilnicki, Li-Jie Jin, and Ming-Chien Shan. An Open, Flexible, and Configurable System for Service Composition. In *Proceedings of the Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000)*, WECWIS '00, pages 125–, Washington, DC, USA, 2000. IEEE Computer Society. 135
46. Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5): 704–739, 1995. 16
47. Stefano Ceri, Marco Brambilla, and Piero Fraternali. The History of WebML Lessons Learned from 10 Years of Model-Driven Development of Web Applications. In Alexander Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, editors, Conceptual Modeling: Foundations and Applications, volume 5600 of *Lecture Notes in Computer Science*, pages 273–292. Springer Berlin/ Heidelberg, 2009. 114
48. Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2002. 113
49. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, March 2001. 58
50. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. World Wide Web Consortium, Note NOTE-wsdl-20010315, March 2001. 81
51. Tyler Close. Decentralized identification, 2001. 17
52. Tyler Close. ACLs don't. Technical Report HPL-2009-20, HP Laboratories, 2009. 17
53. Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2000. 120
54. Walter Colitti, Kris Steenhaut, Niccolo De Caro. Integrating Wireless Sensor Networks with the Web. In *IPSN, USA*, 2011.
55. Melvin E Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968. 20
56. Duncan Cragg. FOREST: An Interacting Object Web. In Erik Wilde and Cesare Pautasso, editors, *REST: From Research to Practice*, pages 161–195. Springer New York, 2011. 40
57. Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). IETF Request for Comments, July 2006. 164
58. Francisco Curbera, Matthew Duftler, Rania Khalaf, and Douglas Lovell. Bite: Workflow Composition for the Web. In *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, pages 94–106, Berlin, Heidelberg, 2007. Springer-Verlag. 135, 136
59. Mathieu D'aquin, Marta Sabou, Enrico Motta, Sofia Angeletou, Laurian Gridinoc, Vanessa Lopez, and Fouad Zablith. What can be done with the Semantic Web? An Overview of Watson-based Applications. In *5th Workshop on Semantic Web Applications and Perspectives*, 2008. 71, 72
60. Doug Davis. Intel Inside Becomes Intel Everywhere, March 2009. http://g.mamund.com/nxmnj. 76
61. Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966. 17

62. Michele Melchiori Devis Bianchini, Valeria Antonellis. Flexible semantic-based service matchmaking and discovery. In *Proceedings international conference on World Wide Web*, volume 11, pages 227–251, 2008. 63
63. Lisa Dusseault and James M. Snell. PATCH Method for HTTP. Internet RFC 5789, March 2010. 46, 53
64. Adam DuVander. 7,000 APIs: Twice as Many as This Time Last Year. ProgrammableWeb, August 2012. 67, 73
65. Justin Ryan, Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. From Representations to Computations: The Evolution of Web Architectures. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, Dubrovnik, Croatia, September 2007. ACM Press. 4, 16
66. Justin Ryan Erenkrantz. Computational REST: A New Model for Decentralized, Internet-Scale Applications DISSERTATION. PhD thesis, University of California, Irvine, September 2009. 4, 16
67. Justin Ryan Erenkrantz, Michael M. Gorlick, Girish Suryanarayana, and Richard N. Taylor. Harmonizing architectural dissonance in REST-based architectures. *Institute for Software Research, Technical Report UCI-ISR-06-18, University of California, Irvine*, December 2006. 4
68. Justin Ryan Erenkrantz, Michael M. Gorlick, and Richard N. Taylor. Rethinking Web Services from First Principles. In *2nd International Conference on Design Science Research in Information Systems and Technology, Pasadena, California*, May 2007. 16
69. Justin Ryan Erenkrantz, Michael M. Gorlick, and Richard N. Taylor. CREST: A New Model for Decentralized, Internet-Scale Applications. Technical Report UCI-ISR-09-4, Institute for Software Research, University of California, Irvine, september 2009. 4
70. Thomas Erl. *Service-oriented architecture*, volume 8. Prentice Hall, New York, 2005. 5
71. Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 186
72. Dave Evans. The Internet of Things: How the Next Evolution of the Internet is Changing Everything, 2011. 20
73. Joel Farrell and Holger Lausen. Semantic Annotations for WSDL and XML Schema (SAWSDL). W3C Recommendation, August 2007. 60
74. Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190. ACM, 1988. 6
75. Ian Fette and Alexey Melnikov. The WebSocket Protocol http://www.ietf.org/rfc/rfc6455.txt. December 2011. 41
76. Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000. 79, 80, 85, 132, 135, 158, 159, 161, 162, 186
77. Roy Thomas Fielding. A Little REST and Relaxation. In *ApacheCon*, November 2008. http://www.slideshare.net/royfielding/a-little-rest-and-relaxation. 86
78. Roy Thomas Fielding. REST APIs must be hypertext-driven. Untangled—Musings of Roy T. Fielding, October 2008. 58, 66, 67
79. Roy Thomas Fielding. REST APIs must be hypertext-driven. Retrieved February 15, 2013, from http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008. 168, 169
80. Roy Thomas Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. Internet RFC 2616, June 1999. 32, 41, 43, 58
81. Roy Thomas Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002. xv, 4, 5, 39, 41, 52
82. Florian F. Fischer and Barry Norton. D3.4.6 microWSMO v2– defining the second version of microWSMO as a systematic approach for rich tagging. Soa4all project deliverable, 2010. 63

83. Brad Fitzpatrick, Brett Slatkin and Martin Atkins. PubSubHubbub Core 0.3, February 2010. 40
84. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Reading, Massachusetts, November 2002. 103
85. Brian Frank, Zach Shelby, Klaus Hartke, and Carsten Bormann. Constrained Application Protocol (CoAP), 2012. 22, 28
86. Ned Freed and Nathaniel S. Borenstein. Multipurpose Internet Mail Extensions (MIME)—Part Two: Media Types. Internet RFC 2046, November 1996. 45
87. Christian Fritz, Richard Hull, and Jianwen Su. Automatic construction of simple artifact-based business processes. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 225–238, New York, NY, USA, 2009. ACM. 135
88. Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An Introduction to UML Profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5–13, 2004. 115
89. Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998. 4, 5
90. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, January 1995. 30, 41, 102
91. Damian D.G. Gessler, Gary S. Schiltz, Greg D. May, Shulamit Avraham, Christopher D. Town, David Grant and Rex T Nelson. SSWAP: A simple semantic Webarchitecture and protocol for semantic Webservices. *BMC Bioinformatics*, 10:309, 2009. 63
92. James J. Gibson. The Ecological Approach to Visual Perception. Psychology Press, 1986. 83
93. Aman Goel, Craig A. Knoblock, and Kristina Lerman. Exploiting Structure within Data for Accurate Labeling Using Conditional Random Fields. In *Proceedings of the 14th International Conference on Artificial Intelligence (ICAI)*, 2012. 69
94. Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983. http://users.ipa.net/dwighth/smalltalk/bluebook/bluebook_imp_toc.html—geprüft: 16. Dezember 2002. 85
95. Michael M. Gorlick, Samuel D. Gasster, Grace S. Peng, and Michael McAtee. Flow Webs: Architecture and Mechanism for Sensor Webs. In *Proceedings of the Ground Systems Architecture Workshop, Manhattan Beach, California, USA, March 26–29 2007*, 2007. 4
96. Michael M. Gorlick, Kyle Strasser, Alegria Baquero, and Richard N Taylor. CREST: principled foundations for decentralized systems. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (SPLASH'11), Companion volume*, pages 193–194. ACM, October 2011. 16
97. Michael M. Gorlick, Kyle Strasser, and Richard N. Taylor. COAST: An architectural style for decentralized on-demand tailored services. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 71–80. IEEE, 2012. 4, 5, 12
98. Joe Gregorio and Bill de Hóra. The Atom Publishing Protocol. Internet RFC 5023, October 2007. 165
99. Joe Gregorio, Roy Thomas Fielding, Marc Hadley, Mark Nottingham, and David Orchard. URI Template. Internet RFC 6570, March 2012. 67
100. Lev Grossman. Apple's New Calling: The iPhone. *Time Magazine*, January 2007. 75
101. Abril Group. Abril Group Sustainability Report. Retrieved November 5, 2012, from http://www.grupoabril.com.br/arquivo/sustainability_report.pdf, 2011. 150
102. Abril Group. Abril Group Institutional Page. Retrieved November 5, 2012, from http://www.grupoabril.com.br/IN/index.shtml, 2012. 150
103. Marc Hadley. Web Application Description Language (WADL). Technical Report TR-2006-153, Sun Microsystems, April 2006. 81
104. Marc Hadley. Web Application Description Language. World Wide Web Consortium, Member Submission SUBM-wadl-20090831, August 2009. 57, 59, 74

105. Brent Hailpern and Peri Lynn Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006. 112
106. David A Halls. *Applying mobile code to distributed systems*. PhD thesis, University of Cambridge, June 1997. 16
107. Norm Hardy. The Confused Deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988. 17
108. Klaus Hartke. Observing Resources in CoAP, 2012. 30
109. Petra Heinl, Stefan Horn, Stefan Jablonski, Jens Neeb, Katrin Stein, and Michael Teschke. A comprehensive approach to flexibility in workflow management systems. *SIGSOFT Softw. Eng. Notes*, 24(2):79–88, March 1999. 135
110. Ian Hickson. Server-Sent Events. World Wide Web Consortium, Candidate Recommendation CR-eventsource-20121211, December 2012. 41
111. Richard Hull. Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems, OTM '08, pages 1152–1163, Berlin, Heidelberg, 2008. Springer-Verlag. 135
112. Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One. W3C recommendation, W3C, December 2004. http://www.w3.org/TR/2004/REC-webarch-20041-215/. 43
113. Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):456–492, 1994. 6
114. Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, February 1988. 16
115. Michael Kaminsky and Eric Banks. SFS-HTTP: Securing the web with self-certifying URLs, 1999. 16
116. Shmuel Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, April 1993. 136
117. Alan C. Kay. The Early History of Smalltalk. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming Languages—II*, pages 511–598. ACM, New York, NY, USA, 1996. 186
118. Gregg Kellogg, Markus Lanthaler, and Manu Sporny. JSON-LD 1.0 Processing Algorithms and API. World Wide Web Consortium, Proposed Recommendation PR-json-ld-api-20131105, November 2013.
119. Rohit Khare and Tantek Çelik. Microformats: A Pragmatic Path to the Semantic Web. In *15th International World Wide Web Conference Posters*, Edinburgh, UK, May 2006. ACM Press. 59
120. Rohit Khare and Richard N. Taylor. Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 428–437, Washington, DC, USA, 2004. IEEE Computer Society. 41
121. Paul Kinlan. WebIntents. https://github.com/PaulKinlan/WebIntents. 90
122. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. 112
123. Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, February 2004. 65
124. Craig Knoblock, Pedro Szekely, Jos Luis Ambite, Aman Goel, Shubham Gupta, Kristina Lerman, Maria Muslea, Mohsen Taheriyan, and Parag Mallick. Semi-Automatically Mapping Structured Sources into the Semantic Web. In *Proceedings of the 9th Extended Semantic Web Conference (ESWC)*, 2012. 68, 69
125. Donald E. Knuth. Literate Programming. *Computer Journal*, 27(2):97–111, 1984. 57
126. Nora Koch. Transformation techniques in the model-driven development process of UWE. In *Workshop proceedings of the sixth international conference on Web engineering*, ICWE '06, New York, NY, USA, 2006. ACM. 113

127. Nora Koch and Andreas Kraus. Towards a common metamodel for the development of web applications. In *Proceedings of the 2003 international conference on Web engineering*, ICWE'03, pages 497–506, Berlin, Heidelberg, 2003. Springer-Verlag. 113
128. J. Kopecký and T. Vitvar. microWSMO. CMS Working Draft, February 2008. 60, 62
129. Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. HRESTS: An HTML Microformat for Describing RESTful Web Services. *In Proceedings of the 2008 IEEE/ACMInternational Conference on Web Intelligence (WI-08)*, 2008. 59, 61, 62
130. Alfred Korzybski. *Science and Sanity*. Colonial Press, 1993. 84
131. David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent Tcl: Targeting the needs of mobile computers. *Internet Computing, IEEE*, 1(4):58–67, 1997. 16
132. Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007. 4
133. Christian Kroib and Nora Koch. UWE Metamodel and Profile: User Guide and Reference. Technical Report 0802, 2008., 2008. 113
134. Reto Krummenacher, Barry Norton, and Adrian Marte. Towards Linked Open Services. In *FIS*, 2010. 64
135. John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289, 2001. 69
136. Markus Lanthaler. Hydra Core Vocabulary Specification. Retrieved from http://www.markus-lanthaler.com/hydra/, 2013. 101
137. Markus Lanthaler and Christian Gütl. A semantic description language for RESTfulData Services to combat Semaphobia. In *Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference*, pages 47–53, June 2011. 66, 67
138. Markus Lanthaler and Christian Gütl. A semantic description language for RESTful data services to combat Semaphobia. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, pages 47–53. IEEE, 2011. 96
139. Markus Lanthaler and Christian Gütl. On using JSON-LD to create evolvable RESTful services. In *Proceedings of the Third International Workshop on RESTfulDesign*, pages 25–32, New York, NY, USA, 2012. ACM. 67
140. Markus Lanthaler and Christian Gütl. Seamless integration of RESTfulservices into the Web of Data. *Advances in Multimedia*, 2012:1:1–1:14, January 2012. 68
141. Holger Lausen, Axel Polleres, and Dumitru Roman. Web Service Modeling Ontology (WSMO). W3C Member Submission, June 2005. 67
142. Paul J. Leach, Michael Mealling, and Richard Salz. A Universally Unique IDentifier (UUID) URN Namespace. Internet RFC 4122, July 2005. 15
143. Ning Li, Carlos Pedrinaci, Maria Maleshkova, Jacek Kopecký, and John Domingue. Omnivoke: A framework for automating the invocation of WebAPIs. In *Proceedings of Fifth IEEE International Conference on Semantic Computing*, pages 380–387, 2011. 62
144. Shitao Li, Jeroen Hoebeke, and Antonio J Jara. Conditional observe in CoAP, 2012. 31
145. Tim Lindholm and Frank Yellin. *The Java virtual machine specification*, volume 297. Addison-Wesley Reading, 1997. 16
146. Daniel Lucrédio. *Uma Abordagem Orientada a Modelos para Reutilização de Software*. PhD thesis, Universidade de São Paulo, 2009. 112
147. Mehmet Ersue, Dan Romascanu, and Juergen Schoenwaelder. Management of Networks with Constrained Devices: Use Cases and requirements, 2012. 22. IETF draft, exp. April 28, 2014.
148. General Magic. Telescript Language Reference. *General Magic, Sunnyvale, California, USA*, October 1995. 16

149. Maria Maleshkova, Carlos Pedrinaci, John Domingue, Guillermo Alvaro Rey, Ivan Martinez. Using Semantics for Automating the Authentication of WebAPIs. *International Semantic Web Conference (ISWC)*, 2010. Shanghai, China. 62, 63, 74
150. Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *Proc. of the 8th IEEE European Conference on Web Services*, 2010. 57
151. Michael Mandel. Where are the Jobs: The App Economy. Technical report, South Mountain Economics, LLC, February 2012. 76
152. Ioana Manolescu, Marco Brambilla, Stefano Ceri, Sara Comai, and Piero Fraternali. Model-driven design and deployment of service-enabled web applications. *ACM Trans. Internet Technol.*, 5:439–479, August 2005. 114
153. David Martin, Mark Burstein, Jerry Hobbs, and Ora Lassila. OWLs: Semantic Markup for Web Services. 65, 67
154. Christian Mayerl, Tobias Vogel, and Sebastian Abeck. SOA-Based Integration of IT Service Management Applications. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 785–786, Washington, DC, USA, 2005. IEEE Computer Society. 111
155. David Mazieres, Michael Kaminsky, M Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. *ACM SIGOPS Operating Systems Review*, 33(5):124–139, 1999. 15, 16
156. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language. W3C Recommendation, February 2004. 65
157. Terry Merriman. MDA in Action: An Anatomy of a Platform-Independent Model, 2003. 123
158. Ali Mesbah and Arie van Deursen. An Architectural Style for Ajax. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society. 41
159. Mark S Miller and Jonathan S Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Eighth Asian Computing Science Conference (ASIAN'03). Programming Languages and Distributed Computation*, pages 224–242. Springer, 2003. 17
160. Mark Samuel Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 16, 17
161. Richard Morris. Roy Fielding: Geek of the Week, August 2010. https://www.simple-talk.com/opinion/geek-of-the-week/roy-fielding-geek-of-the-week/. 80
162. Toby Murray. *Analysing the security properties of object-capability patterns*. PhD thesis, Hertford College, University of Oxford, Oxford, UK, 2010. 17
163. Toby Murray and Duncan Grove. Non-delegatable authorities in capability systems. *Journal of Computer Security*, 16(6):743–759, 2008. 17
164. Bruce Jay Nelson. *Remote procedure call*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1981. 5
165. Microsoft Software Developer Network. Using HTTP as an RPC Transport, September 2011. http://g.mamund.com/vxsoc. 79
166. Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, July 2003. 135
167. Donald Norman. *The Design of Everyday Things*. Basic Books, September 2002. 83, 86
168. Barry Norton and Steffen Stadtmüller. Scalable Discovery of Linked Services. In *RED*, 2011. 64
169. Mark Nottingham. Web Linking. Internet RFC 5988, October 2010. 43
170. Mark Nottingham. Linking in JSON. Retrieved November 5, 2012, from http://www.mnot.net/blog/2011/11/25/linking_in_json, 2011. 166
171. Mark Nottingham, Julian Reschke, and Jan Algermissen. Link Relations. Retrieved November 5, 2012, from http://www.iana.org/assignments/link-relations/link-relations.xml, 2012. 40, 165

172. Mark Nottingham and Robert Sayre. The Atom Syndication Format. Internet RFC 4287, December 2005. 40, 165
173. Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. 8, 14
174. OMG. MDA Guide Version 1.0.1, 2003. 114
175. OMG. UML Profile for Enterprise Application Integration (EAI), 2004. 113
176. OMG. MOF 2.0/XMI Mapping, v2.1.1, 2007. 122
177. OMG. UML Profile for Modeling and Analysis of Real-time and Embedded Systems, 2009. 113
178. Oracle. Your First Cup: An Introduction to the Java EE Platform, 2012. 115
179. Savas Parastatidis, Jim Webber, Guilherme Silveira, and Ian Robinson. The Role of Hypermedia in Distributed System Development. In Pautasso et al. [183], pages 16–22. 94
180. Cesare Pautasso. BPEL for REST. In *Proceedings of the 6th International Conference on Business Process Management*, BPM '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. 135
181. Cesare Pautasso. Composing RESTful Services with JOpera. In *Proceedings of the 8th International Conference on Software Composition*, SC '09, pages 142–159, Berlin, Heidelberg, 2009. Springer-Verlag. 135
182. Cesare Pautasso, Erik Wilde, and Rosa Alarcón, editors. *Second International Workshop on RESTful Design (WS-REST 2011)*, Hyderabad, India, March 2011. xvii, 199
183. Cesare Pautasso, Erik Wilde, and Alexandros Marinos, editors. *First International Workshop on RESTful Design (WS-REST 2010)*, Raleigh, North Carolina, April 2010. xvii, 198
184. Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *17th International World Wide Web Conference*, pages 805–814, Beijing, China, April 2008. ACM Press. 111
185. Carlos Pedrinaci and John Domingue. Toward the Next Wave of Services: Linked Services for the Web of Data. *Journal of Universal Computer Science*, pages 1694–1719, 2010. 62
186. Carlos Pedrinaci, Dong Liu, Maria Maleshkova, David Lambert, Jacek Kopecky, John Domingue. iServe: A Linked Services Publishing Platform. *Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference*, 2010. 62, 63, 72
187. Carlos Pedrinaci and John Domingue. Toward the Next Wave of Services: Linked Services for the Web of Data. *Journal of Universal Computer Science*, 16(13):1694–1719, July 2010. 70
188. Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, October 2003. 135
189. Adrien Piérard and Marc Feeley. Towards a portable and mobile Scheme interpreter. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 59–68, 2007. 16
190. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The Use of Name Space in Plan 9. *Operating Systems Review*, 27(2):72–76, April 1993. 188
191. Dave Presotto and Phil Winterbottom. The Organization of Networks in Plan 9. In *USENIX Conference*, pages 271–280, 1993. 188
192. Matthias Quasthoff and Christoph Meinel. Semantic Web Admission Free–Obtaining RDF and OWL Data from Application Source Code. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, Karlsruhe, Germany*, pages 17–25, 2008. 104
193. Rajendra K Raj, Ewan Tempero, Henry M Levy, Andrew P Black, Norman C Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software: Practice and Experience*, 21(1):91–118, 1991. 17
194. Jonathan Allen Rees. *A security kernel based on the Lambda-Calculus*. PhD thesis, Massachusetts Institute of Technology, 1996. 16
195. Machina Research. The Connected Life: A USD4.5 trillion global impact in 2020, 2012. 20

196. Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2007. 111
197. Adam Roach. A SIP Event Package for Subscribing to Changes to an HTTP Resource. RFC 5989 (proposed standard), IETF, August 2010. http://www.ietf.org/rfc/rfc5989.txt. 43
198. Doug Rosenberg and Matt Stephens. *Use Case Driven Object Modeling with UML*. Apress, 2007. 120
199. Florian Rosenberg, Francisco Curbera, Matthew J. Duftler, and Rania Khalaf. Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. *IEEE Internet Computing*, 12(5):24–31, 2008. 136
200. David S. Rosenblum and Alexander L. Wolf. A design framework for Internet-scale event observation and notification. In Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC '97/FSE-5, pages 344–360, New York, NY, USA, 1997. Springer-Verlag New York, Inc. 41
201. RSS Advisory Board. Really Simple Syndication 2.0, March 2009. 40
202. Jerome H Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974. 6
203. Andrea Schauerhuber, Manuel Wimmer, and Elisabeth Kapsammer. Bridging existing Web modeling languages to model-driven engineering: a metamodel for WebML. In *Workshop proceedings of the sixth international conference on Web engineering*, ICWE '06, New York, NY, USA, 2006. ACM. 112
204. Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil M. P. van der Aalst. Process Flexibility: A Survey of Contemporary Approaches. In Jan L. G. Dietz, Antonia Albani, and Joseph Barjis, editors, *CIAO!/EOMAS*, volume 10 of *Lecture Notes in Business Information Processing*, pages 16–30. Springer, 2008. 135
205. Silvia Schreier. Modeling RESTful Applications. In Pautasso et al. [182], pages 15–21. 113
206. Nelly Schuster, Christian Zirpins, and Ulrich Scholten. How to balance flexibility and coordination? Service-oriented model and architecture for document-based collaboration on the Web. In *SOCA*, pages 1–9, 2011. 136
207. Charles Severance. Discovering JavaScript Object Notation. *Computer*, 45(4):6–8, April 2012. 67
208. Jonathan Strauss Shapiro. *EROS: A capability system*. PhD thesis, University of Pennsylvania, Philadelphia, USA, 1999. 16, 17
209. Zach Shelby. CoRE Link Format, 2012. 32
210. Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley, 2009. 20
211. Amit P. Sheth, Kunal Verma, and Karthik Gomadam. Semantics to Energize the Full Services Spectrum. *Comm. ACM*, 49:55–61, 2006. 61
212. Amit P. Sheth, Karthik Gomadam, Jon Lathem. SAREST: Semantically Interoperable and Easier-to-Use Services and Mashups. *IEEE Internet Computing*, 11(6):91–94, 2007. 61, 62
213. Elizabeth Shogren. Don't Trash Or Stash Old Cell Phones; Recycle Them. *NPR*, April 2010. http://www.npr.org/templates/story/story.php?storyId = 125657764. 82
214. sitemaps.org. Sitemaps XML format. 0.90, Google, February 2008. http://www.sitemaps.org/protocol.php. 40
215. Carlos A. Soto. 5 technologies that will change the market, August 2010. 67
216. Steve Speicher, John Arwe, and Ashok Malhotra. Linked Data Platform 1.0. World Wide Web Consortium, Working Draft WD-ldp-20130730, July 2013. 108
217. Sebastian Speiser and Andreas Harth. Integrating Linked Data and Services with Linked Data Services. In *The Semantic Web: Research and Applications*, volume 6643 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2011. 64
218. Manu Sporny, Gregg Kellogg, and Markus Lanthaler. JSON-LD 1.0: A JSON-based Serialization for Linked Data. World Wide Web Consortium, Proposed Recommendation PR-json-ld-20131105, November 2013.

219. James W. Stamos and David K. Gifford. Implementing remote evaluation. *Software Engineering, IEEE Transactions on*, 16(7):710–722, 1990. 16
220. James W. Stamos and David K Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):537–564, 1990. 5, 16
221. James W. Stamos. *Remote evaluation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1986. 5
222. Christoph Szymanski and Silvia Schreier. Case Study: Extracting a Resource Model from an Object-Oriented Legacy Application. In Alarcón et al. [5], pages 19–24. 113
223. Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and Jose Luis Ambite. Rapidly Integrating Services into the Linked Data Cloud. In *Proceedings of the 11th International Semantic Web Conference (ISWC)*, Boston, USA, 2012. 69
224. Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and Jose Luis Ambite. Semi-Automatically Modeling Web APIs to Create Linked APIs. In *Proceedings of the Linked APIs for the Semantic Web Workshop (LAPIs)*, Heraklion, Crete, Greece, 2012. 70
225. Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the 41st IEEE International Computer Conference (COMPCON'96), WA DC, USA*, pages 58–63. IEEE Computer Society, 1996. 16
226. Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Trans. Softw. Eng.*, 22(6):390–406, June 1996. 79
227. OASIS Web Services Business Process Execution Language (WSBPEL) TC. Business Process Execution Language for Web Services (BPEL) version 2.0, 2007. 133
228. Amis Technology. Model Driven Architecture (MDA), 2006. 119
229. Amit P. Sheth, Karthik Gomadam, Jon Lathem. Challenges in Deployment of Model Driven Development. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 15–20, 2009. 112
230. Dave Thomas and Brian M. Barry. Model driven development: the case for domain oriented programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 2–7, New York, NY, USA, 2003. ACM. 113
231. Sebastian Thöne, Ralph Depke, and Gregor Engels. Process-Oriented, Flexible Composition of Web Services with UML. In *ER (Workshops)*, pages 390–401, 2002. 135
232. Rattapoom Tuchinda, Craig A. Knoblock, and Pedro Szekely. Building Mashups by Demonstration. *ACMTransactions on the Web (TWEB)*, 5(3), 2011. 68
233. Sameer Tyagi. RESTful Web Services. Technical report, Oracle Technology Network, August 2006. http://www.oracle.com/technetwork/articles/javase/index-137171.html. 79
234. Francisco Valverde and Oscar Pastor. Dealing with REST Services in Model-driven Web Engineering Methods. *V Jornadas Cientfico-Tcnicas en Servicios Web y SOA, JSWEB*, 2009. 113, 114
235. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000. 113
236. Anne van Kesteren. Cross-Origin Resource Sharing. W3C Working Draft, April 2012. 73
237. Adam Du Vander. 4,000 Web APIs: What's Hot and What's Next?, October 2011. http://blog.programmableweb.com/2011/10/03/4000-web-apis-whats-hot-and-whats-next/. 77
238. Ruben Verborgh, Vincent Haerinck, Thomas Steiner, Davy Van Deursen, Sofie Van Hoecke, Jos De Roo, Rik Van De Walle, and Joaquim Gabarró Vallés. Functional Composition of Sensor Web APIs. In *Proceedings of the 5th International Workshop on Semantic Sensor Networks*, November 2012. 66
239. Ruben Verborgh, Thomas Steiner, Joaquim Gabarró Vallés, Erik Mannens, and Rik Van de Walle. A Social Description Revolution—Describing Web APIs' Social Parameters with RESTdesc. In *Proceedings of the AAAI 2012 Spring Symposia*, March 2012. 66
240. Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Sam Coppens, Joaquim Gabarró Vallés, and Rik Van de Walle. Functional Descriptions as the Bridge between

Hypermedia APIs and the Semantic Web. In *Proceedings of the Third International Workshop on RESTfulDesign*. ACM, April 2012. 64, 65, 66
241. Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Jos De Roo, Rik Van de Walle, and Joaquim Gabarró Vallés. Capturing the functionality of Web services with functional descriptions. *Multimedia Tools and Applications*, 2013. 65
242. Hans Vestberg. CEO to shareholders: 50 billion connections 2020. Press Release, April 13, 2010, http://www.ericsson.com/thecompany/press/releases/2010/04/1403231. 76
243. Steve Vinoski. Serendipitous Reuse. *IEEE Internet Computing*, 12(1):84–87, January 2008. 94
244. Paul de Vrieze, Lai Xu, Athman Bouguettaya, Jian Yang, and Jinjun Chen. Process-Oriented Enterprise Mashups. In *Proceedings of the 2009 Workshops at the Grid and Pervasive Computing Conference*, GPC '09, pages 64–71, Washington, DC, USA, 2009. IEEE Computer Society. 135
245. Dimitris Vyzovitis and Andrew Lippman. MAST: A dynamic language for programmable networks. Technical report, MIT Media Laboratory, May 2002. 16
246. Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SIGOPS)*, volume 27:5, pages 203–216. ACM, December 1993. 16
247. Jim Webber. REST in Practice: Hypermedia and Systems Architecture. O'Reilly, 2010. 57
248. Adam Wick and Matthew Flatt. Memory accounting without partitions. In *Proceedings of the 4th international symposium on Memory management*, pages 120–130. ACM, October 2004. 16
249. Erik Wilde. REST and RDF Granularity, May 2009. 64
250. Erik Wilde and Cesare Pautasso, editors. *REST: From Research to Practice*. Springer-Verlag, New York, NY, 2011. xvii, 191
251. M. D. Wilkinson and M. Links. BioMOBY: An open source biological web services proposal. *Briefings in Bioinformatics*, 3:331–341, 2002. 63
252. Erik Wittern, Nelly Schuster, Jörn Kuhlenkamp, and Stefan Tai. Participatory Service Design through Composed and Coordinated Service Feature Models. In *Proceedings of the 10th International Conference on Service Oriented Computing*, pages 158–172, 2012. 136
253. Michael Scott Wolfe. SCURL authentication: A decentralized approach to entity authentication. Master's thesis, University of California, Irvine, October 2011. 15
254. WS Description Working Group. Web Service Description Language (WSDL) ersion 2.0, W3C Proposed Recommendation, May 2007. 58
255. Bo Wu, Pedro Szekely, and Craig A. Knoblock. Learning Data Transformation Rules through Examples: Preliminary Results. In *9th International Workshop on Information Integration on the Web (IIWeb)*, 2012. 68
256. Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53:91–99, January 2010. 16
257. Aydan R Yumerefendi and Jeffrey S Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, volume 5, pages 3–3, 2005. 5
258. Kris Zyp. A JSON Media Type for Describing the Structure and Meaning of JSON Documents. Internet Draft draft-zyp-json-schema-01, December 2009. 67