RESTful Web services: principles, patterns, emerging technologies

Cesare Pautasso

Abstract RESTful Web services are software services which are published on the Web, taking full advantage and making correct use of the HTTP protocol. This chapter gives an introduction to the REST architectural style and how it can be used to design Web service APIs. We summarize the main design constraints of the REST architectural style and discuss how they impact the design of so-called RESTful Web service APIs. We give examples on how the Web can be seen as a novel kind of software connector, which enables the coordination of distributed, stateful and autonomous software services. We conclude the chapter with a critical overview of a set of emerging technologies which can be used to support the development and operation of RESTful Web services.

1 Introduction

REST stands for REpresentational State Transfer [13]. It is the architectural style that explains the quality attributes of the World Wide Web, seen as an open, distributed and decentralized hypermedia application, which has scaled from a few Web pages in 1990 up to billions of addressable Web resources today [6, 4]. Even if it is no longer practical to take a global snapshot of the Web architecture, seen as a large set of Web browsers, Web servers, and their collective state, it is nevertheless possible to describe the style followed by such Web architecture. The REST architectural style includes the design constraints which have been followed to define the HTTP protocol [12], the fundamental standard together with URI and HTML which has enabled to

Cesare Pautasso
Faculty of Informatics, University of Lugano, via Buffi 13, CH-6900, Lugano, Switzerland, e-mail: c.pautasso@ieee.org

build the Web [5]. These constraints make up the REST architectural style and have been distilled by Roy Fielding in his PhD dissertation [11].

Over the last decade, the Web has grown from a large-scale hypermedia application for publishing and discovering documents (i.e., Web pages) into a programmable medium for sharing data and accessing remote software components delivered as a service. As the Web became widespread, TCP/IP port 80 started to be left open by default on most Internet firewalls, making it possible to use the HTTP protocol [12] (which by default runs on port 80) as a universal mean for tunneling messages in business to business integration scenarios. RESTful Web services — as opposed to plain (or Big [22]) Web services — emphasize the correct and complete use of the HTTP protocol to publish software systems on the Web [24]. More and more services published on the Web are claiming to be designed using REST. As we are going to discuss, even if all make use of the HTTP protocol natively, not all of them do so in full compliance with the constraints of the REST architectural style [16].

In this chapter we present how the Web can be seen as a novel kind of software connector, which enables the coordination of distributed, stateful and autonomous software services. We summarize the main design constraints of the REST architectural style and discuss how they impact the design of so-called RESTful Web service APIs. We conclude the chapter with a critical overview of a set of emerging technologies which can be used to support the development and operation of RESTful Web services.

2 Principles

Understanding the architectural principles underlying the World Wide Web can lead to improving the design of other distributed systems, such as integrated enterprise architectures. This is the claim of RESTful Web services, designed following the REST architectural style [11], which emphasizes the scalability of component interactions, promotes the reuse and generality of component interfaces, reduces coupling between components, and makes use of intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

2.1 Design Constraints

The main design constraints of the REST architectural style are: global addressability through resource identification, uniform interface shared by all resources, stateless interactions between services, self-describing messages, and hypermedia as a mechanism for decentralized resource discovery by referral.

1. *Addressability* All resources that are published by a Web service should be given a unique and stable identifier [17]. 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 concept of a resource is kept very general as REST intentionally does not make any assumptions on the corresponding implementation. A resource can be used to publish some service capability, a view over the internal state of a service, as well as any source of machine-processable data, which may also include meta-data about the service.
2. *Uniform Interface* All resources interact through a uniform interface, which provides a small, generic and functionally sufficient set of methods to support all possible interactions between services. Each method has a well defined semantics in terms of its effect on the state of the resource. In the context of the Web and its HTTP protocol, the uniform interface comprises the methods (e.g., GET, PUT, DELETE, POST, HEAD, OPTIONS, etc.) that 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., PATCH has been recently proposed as an addition to deal with partial resource updates [8]) and other protocols based on HTTP such as WebDAV include additional methods [14]
3. *Stateless Interactions* Services do not establish any permanent session between them which spans across more than a single interaction. This ensures that requests to a resource are independent from each other. At the end of every interaction, there is no shared state that remains between clients and servers. Requests may result in a state change of the resource, whose new state becomes immediately visible to all of its clients.
4. *Self-Describing Messages* Services interact by exchanging request and response messages, which contain both the data (or the representations of resources) and the corresponding meta-data. Representations can vary according to the client context, interests and abilities. For example, a mobile client can retrieve a low-bandwidth representation of a resource. Likewise, a Web browser can request a representation of a Web page in a particular language, according to its user preferences. This greatly enhances the degree of intrinsic interoperability of a REST architecture, since a client may dynamically negotiate the most appropriate representation format (also called media type) with the resource as opposed to forcing all clients and all resources to use the same format. Request and response messages also should contain explicit meta-data about the representation so that services do not need to assume any kind of out-of-band agreement on how the representation should be parsed, processed and understood.
5. *Hypermedia* Resources may be related to each other. Hypermedia is about embedding references to related resources inside resource representations or in the corresponding meta-data. Clients can thus discover the identifiers (or hyper-links) of related resources when processing representations and choose to follow the link as they navigate the graph built out of rela-

tionships 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" [3].

2.2 Maturity Model

The main design constraints of the REST architectural style can also be adopted incrementally, leading to the definition of a maturity model for RESTful Web services as proposed by Leonard Richardson. This has led to a discussion on whether only services that are fully mature can be actually called RESTful. In the state of the practice, however, many services which are classified in the lower levels of maturity already present themselves as making use of REST.

- *Level 0: HTTP as a tunnel* These are all services which simply exchange XML documents (sometimes referred to as Plain-Old-XML documents as opposed to SOAP messages) over HTTP POST request and responses, effectively following some kind of XML-RPC protocol [28]. A similar approach is followed by services which replace the XML payloads with JSON, YAML or other formats which are used to serialize the input and output parameters of a remote procedure call, which happens to be tunneled through an open HTTP endpoint. Even if such services are not making use of SOAP messages, they are not really making full use of the HTTP protocol according to the REST constraints either. In particular, since all messages go to the same endpoint URL, a service can distinguish between different operations only by parsing such information out of the XML (or JSON) payload.
- *Level 1: Resources* As opposed to using a single endpoint for tunneling RPC messages through the HTTP protocol, services on maturity level 1 make use of multiple identifiers to distinguish different resources. Each interaction is addressed to a specific resource, which can however still be misused to identify different operations or methods to be performed on the payload, or to identify different instances of object of a given class, to which the request payload is addressed.
- *Level 2: HTTP Verbs* In addition to fine-grained resource identification, services of maturity level 2 also make proper use of the REST uniform interface in general and of the HTTP verbs in particular. This means that not only clients can perform a GET, DELETE, PUT on a resource, in addition to POSTing to it, but also do so in compliance with the semantics of such methods. For example, service designers ensure that GET, PUT and DELETE requests to their service are idempotent. Since we can assume

that the HTTP methods are used according to their standard semantics, we can use the corresponding safety and idempotency properties to optimize the system by introducing intermediaries. For example, the results of safe and side-effect free GET requests can be cached and failed PUT and DELETE requests can be automatically retried. Additionally, services make use of HTTP status codes correctly to, e.g., indicate whether methods are applicable to a given resource or to assign blame between which party is responsible for a failed interaction.

- *Level 3: Hypermedia* These are the fully mature RESTful Web services, which in addition to exposing multiple addressable resources which share the same uniform interface also make use of hypermedia to model relationship between resources. This is achieved by embedding so-called *hypermedia controls* within resource representations [19]. Depending on the chosen media type, hypermedia controls such as links or forms can be parsed, recognized and interpreted by clients to drive their navigation within the graph of related resources. Hypermedia controls will be typed according to the semantics of the relationship and contain all information necessary for a client to formulate a request to a related resource. As opposed to knowing in advance all the addresses of the resources that will be used, a client can thus dynamically discover with which resource it should interact by following links of a certain type. Key to achieving this level of maturity is the choice of media types which support hypermedia controls (e.g., XML or JSON do not, while ATOM, XHTML or JSON-LD do.). The ability of a service to change the set of links that are given to a client based on the current state of a resource is also known with the ugly HATEOAS (Hypertext As The Engine Of Application State) acronym, to which now the simpler "hypermedia" term is preferred [23].

The maturity level of a service also affects the quality attributes of the architecture in which the service is embedded. Tunneling messages through an open HTTP port (level 0) leads only to the basic ability to communicate and exchange data, but – security issues notwithstanding – is likely to result in brittle integrated systems, which are difficult to evolve and scale. Distinguishing multiple resources helps to apply divide and conquer techniques to the design of a service interface and enable services to use global identifiers to address each resource that is being published. Applying a standardized and uniform interface to each resource removes unnecessary variations (as there are only a few universally accepted methods applicable to a resource) and enables all services to interact with all resources within the architecture, thus promoting interoperability and serendipitous reuse [29]. Additionally the semantics of the methods that make up the uniform interface can be adjusted so that the scalability and reliability of the architecture are enhanced. However, only the dynamic discoverability of resources provided by hypermedia contributes to minimize the coupling within the resulting architecture.

*2.3 Comparing REST vs. WS-**

The maturity model can also be used to give a rough comparison between RESTful Web services and WS-* Web Services (Figure 1). A more detailed comparison can be found in [22].

As the maturity level increases, the service will switch from using a single communication endpoint to many URIs (on the resource identification axis). Likewise, the set of possible methods (or operations) will be limited to the ones of the uniform interface as opposed to designing each service with its own set of operations explicitly described in a WSDL document. From a REST perspective, all WSDL operations are tunneled through a single HTTP verb (POST), thus reducing the expressiveness of HTTP seen as an application protocol, which is used as a transport protocol for tunneling messages. In WSDL several communication endpoints can be associated with the same service although these endpoints are not intended for distinguising HTTP resources but may be used to access the same service through alternative communication mechanisms.

The third axis is not directly reflected in the maturity model but is also important for understanding the difference between the two technology stacks, one having a foundation in the SOAP protocol and the XML format, while the other leaves open the choice of which message format should be used (shown on the representations axis) so that clients and services can negotiate the most suitable format to achieve interoperability.

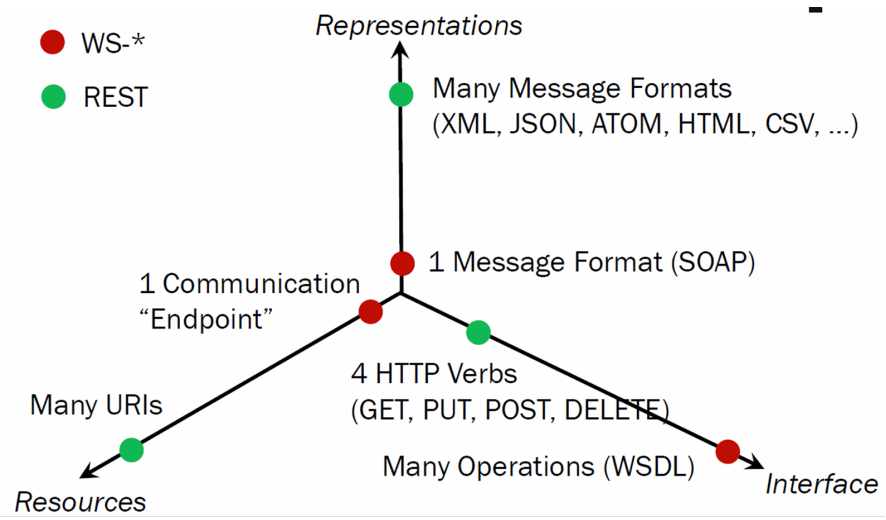

Fig. 1 Design Space: RESTful Web Services vs. WS-* Web Services

3 Example

As inspiration for this example we use the Doodle REST API, which gives programmatic access to the Doodle poll Web service available at (http://www.doodle.ch). Doodle is a very popular service, which allows to minimize the number of emails exchanged in order to find an agreement among a set of people. The service allows to initiate polls by configuring a set of options (which can be a set of dates for scheduling a meeting, but can also be a set of arbitrary strings). The link to the poll is then mailed out to the participants, who are invited to answer the poll by selecting the preferred options. The current state of the poll can be polled at any time by the initiator, who will typically inform the participants of the outcome with a second email message.

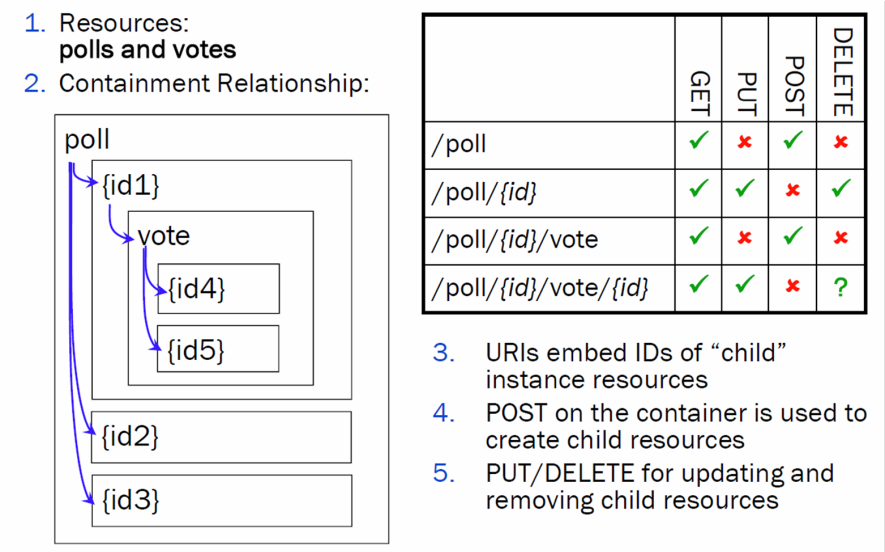

Fig. 2 Simple Doodle REST API

The Simple Doodle REST API (Figure 2) publishes two kinds of resources: polls (a set of options once can choose from) and votes (choices of people within a given poll). There is a natural containment relationship between the two kinds of resources, which fits naturally into the convention to use `/` as a path separator in URIs. Thus the service publishes a `/poll` root resource, which contains a set of `/poll/{id}` poll instances, which include the corresponding set of votes `/poll/{id}/vote/{id}`.

3.1 Listing active polls

The root `/poll` resource is used to retrieve (with GET) the list of links to the polls which have been instantiated:

```
⇒GET /poll
  Accept: text/uri-list

⇐200 OK
  Content-Type: text/uri-list

  http://doodle.api/poll/201204301
  http://doodle.api/poll/201204302
  http://doodle.api/poll/201205011
```

3.2 Creating new polls

The same `/poll` resource acts as factory resource which accepts POST requests to create new poll instances. The identifier of the newly created poll is returned as a link associated with the `Location` response header.

```
⇒POST /poll
  Content-Type: application/xml
  <options>A,B,C</options>

⇐201 Created
  Location: /poll/201205012
```

3.3 Fetching the current state of a poll

The current state of a poll instance can be read with GET, modified with PUT (e.g., to change the set of possible options or to close the poll). Poll instances can also be removed with DELETE.

```
⇒GET /poll/201205012
  Accept: application/xml
```

```
⇐200 OK
  Content-Type: application/xml

  <poll>
  <options>A,B,C</options>
  <votes href="/poll/201205012/vote"/>
  </poll>
```

The representation of a newly created poll resource, in addition to the set of options provided by the client, also contains a link to the resource used to cast votes. Clients can follow the link to express their opinion and make a choice. The nested vote resource acts as a factory resource for individual votes.

3.4 Casting votes

```
⇒POST /poll/201205012/vote
  Content-Type: application/xml
  <vote>
  <name>C. Pautasso</name>
  <choice>B</choice>
  </vote>
⇐201 Created
  Location: /poll/201205012/vote/1
```

After the previous request has been processed a new vote has been cast and the state of the poll has changed. Retrieving it will now return a different representation, which includes the information about the vote.

```
⇒GET /poll/201205012
  Accept: application/xml
⇐200 OK
  Content-Type: application/xml

  <poll>
  <options>A,B,C</options>
  <votes href="/poll/201205012/vote">
  <vote id="1">
  <name>C. Pautasso</name>
  <choice>B</choice>
  </vote>
  </votes>
  </poll>
```

3.5 Changing votes

Since each vote gets its own URI it is also possible to manipulate its state with PUT and DELETE. For example, clients may want to retract a vote (with DELETE) or modify the choice (with PUT) as in the following example.

```
⇒PUT /poll/201205012/vote/1
  Content-Type: application/xml
  <vote>
  <name>C. Pautasso</name>
  <choice>C</choice>
  </vote>

⇐200 OK
```

3.6 Interacting with votes

In general, it is not always possible nor it is necessary for a resource to respond to requests which make use of all possible methods of the uniform interface. In the context of the Simple Doodle REST API, as shown in Figure 2, it has been chosen not to support PUT and DELETE on the `/poll` and `/poll/{id}/vote` resources. Also POST requests to individual instances `/poll/{id}` or `/poll/{id}/vote/{id}` are not supported. Such requests do not have a meaningful effect on the state of the resource and are thus disallowed. Clients attempting to issue them will receive an erroneous response:

```
⇒POST /poll/201205012/vote/1

⇐405 Method not allowed
```

Clients can also inquire which methods are allowed before attempting to perform them on a resource making use of the OPTIONS method as follows

```
⇒OPTIONS /poll/201205012/vote/1

⇐204 No Content
  Allow: GET, PUT, DELETE
```

An OPTIONS request will return a list of the methods which are currently applicable to a resource in the response `Allow` header. The set of allowed methods may change depending on the state of the resource.

3.7 Removing a poll

Once a poll has received enough votes and a decision has been made, its state will be kept indefinitely by the service until an explicit request to remove it is made by a client.

⇒`DELETE /poll/201205012`

⇐`200 OK`

Subsequent requests directed to the delete poll instance will also receive an erroneous response.

⇒`GET /poll/201205012`

⇐`404 Not Found`

4 Patterns

Once the basic architectural principles for the design of RESTful Web services are established, it remains sometimes difficult to apply them directly to the design of specific Web service APIs. In this Section we collect a small number of design patterns, which provide some guidance on how to deal with resource creation, long running operations and concurrent updates. Additional known patterns address features such event notifications, enhancing the reliability of interactions, atomicity and transactions and supporting the evolution of service interfaces. In general, applying one of these patterns requires to make use of some existing feature of the standard HTTP protocol, which may need to be augmented with some conventions and shared assumptions on how to interpret its status code and headers. The current understanding within the REST community is that it should be possible to design fully functional service APIs that do not require any non-standard extension to the HTTP protocol.

The example patterns included in this chapter is not intended to be complete, for additional guidance on how to design RESTful Web services, we refer the interested reader to [1, 7, 10, 25, 30].

4.1 Resource creation

The instantiation of resources is a key feature of most RESTful Web services, which enable clients to create new resource identifiers and set the corresponding state to an initial value. The resource identifier can either be set by the client or by the service. It is easier to guarantee that URIs created by the

service are unique, while it is possible that multiple clients will generate the same identifier.

When using a single HTTP interaction to create a resource, there are two possible verbs that can be used: PUT or POST. The basic semantics of PUT requests is to update the state of the corresponding resource with the provided payload. If no resource is found with the given identifier, a new resource is created. This has the advantage of using idempotent requests to create a resource, but requires clients to avoid mixing up resource identifiers. POST on the other hand assumes that the server will create a new resource identifier. Since POST is not idempotent, there have been a number of patterns that have been proposed to address this limitation and avoid the so-called "duplicated POST submission" problem. The convention is to use some kind of "factory" resource, to which POST requests are directed for creating new resources. However, repeating such requests in case of failure would lead to potentially multiple, different instances to be created by the factory.

The pattern is based on the idea of splitting the centralized generation of the new resource identifier on the service-side from the initialization of its state with the payload provided by the client. The pattern makes combined usage of both POST and PUT requests as follows.

```
⇒POST /factory
  <Empty Payload>

⇐303 See Other
  Location: /factory/id

⇒PUT /factory/id
  <Initialization Payload>

⇐200 OK
```

The first POST request returns a new unique resource identifier `/factory/id` but does not initialize its corresponding resource since the payload is empty. The second request PUTs the initial state on the new resource. In the worst case, failures during the first POST request will lead to lost resource identifiers, which however can be garbage collected by the server since the corresponding resource has not been initialized. Likewise, clients may fail between the two requests and thus could forget to follow up with the PUT request. The designer of the service needs to make reasonable assumptions on the maximum allowed delay between the two interactions. If a client is too late and the resource identifier has been already garbage collected by the server, then another one can be simply retrieved by repeating the first POST request.

Variations of this pattern have been proposed which replace the initial POST with a GET request, which in the same way returns a new unique identifier every time it is invoked. Similarly, the response payload of the first request could be used to provide the client with a representation template, i.e., a form to be completed with the information required to initialize the new resource.

4.2 Long Running Operations

HTTP is a client/server protocol which does not assume that every request is followed by a response indicating that the work has completed. For long running operations, which may result in a timeout of the network communication, it is possible to break the connection and avoid blocking the client for too long. This is particularly useful to invoke service operations that – depending on the size of the input provided by clients or by the complexity of their internal implementation – may require a long time to complete processing it.

The pattern is based on turning the long running operation into a resource, whose identifier can be returned immediately to the client submitting the corresponding job.

```
⇒POST /job
  Input data payload

⇐202 Accepted
  Content-Location: /job/201205019

  <job>
  <status>pending</status>
  <message>Your job has been queued for processing</message>
  <ping-time>2012-05-01T05:22:12Z</ping-time>
  </job>
```

The 202 Accepted status code implies that the service has verified the request input payload and has accepted it, but no immediate response can be given. The client should follow the link given in the `Content-Location` header to inquire (with GET) about the status of the pending request.

```
⇒GET /job/201205019

⇐200 OK
  <job>
  <status>processing</status>
  <message>Your job is being processed</message>
  <ping-time>2012-05-01T06:22:09Z</ping-time>
  </job>
```

Clients can send GET requests to the job resource at any time to track its progress. In addition to the status, the response also contains a hint (in the `ping-time` element) on when the next poll request should be performed in order to reduce network traffic and service load due to excessive polling.

Once the job has been completed, the response to the poll request will redirect the client to another resource from which the final result can be retrieved.

```
⇒GET /job/201205019
```

```
⇐303 See Other
  Location: /job/201205019/output
  <job>
  <status>done</status>
  <message>Your job has been successfully completed</message>
  </job>
```

The client can then follow the link found in the `Location` header to retrieve (with GET) the output of the completed job. The link could also be shared among different clients interested in reading the output of the original POST request.

```
⇒GET /job/201205019/output

⇐200 OK
  Output data payload
```

In case the client is no longer interested in retrieving the results, it is possible to cancel the resource job and thus remove it from the queue of pending requests. The client thus issues a DELETE request on the job resource, which will be allowed as long as the job has not yet completed its execution.

```
⇒DELETE /job/201205019

⇐200 OK
```

After a request has completed it is no longer possible to cancel it. In this case, a similar DELETE request can be performed on the resource representing the output results of the job when the client has completed downloading them and it is no longer interested in keeping the results stored on the server.

```
⇒DELETE /job/201205019/output

⇐200 OK
```

If clients do not remember to clean up after themselves the server can end up storing a copy of all long running requests and potentially run out of space. Still, a garbage collection mechanism can be implemented to automatically remove old results through the same DELETE request.

This pattern shows how to deal with long running operations by applying a general design principle of turning "everything into a resource" [24]. In this case the resource represents the long running request which is managed by the client through the HTTP uniform interface.

4.3 Optimistic Locking

RESTful Web services are stateful services, which associate to each resource URI a representation which is produced based on the current state of the corresponding resource. It is thus important to deal with concurrent state modifications without violating the stateless constraint, which prevents clients to establish a session with a service in which the resource is updated. The problem addressed by this pattern is thus the one of dealing with concurrent resource updates in compliance with the stateless constraint. The solution adopted by the HTTP protocol makes use of a form of optimistic locking, as follows.

1. The client retrieves the current state of a resource.
   ⇒`GET /resource`

   ⇐`200 OK`
   `ETag: 1`

   `Current representation`
   Together with the representation of the resource, the client is given through the `ETag` header some meta-data which identifies the current version of the resource.
2. The client updates the state of a resource. While doing so, the client uses the `If-Match` header to make the request conditional.
   ⇒`PUT /resource`
   `If-Match: 1`

   `New representation`

   ⇐`200 OK`
   `ETag: 2`

   `Updated representation`

The server will execute the `PUT` request only if the version of the resource (on the server-side) matches the version provided within the client request. If there is a mismatch, another client has already updated the resource in the meanwhile and an update conflict has been detected. This is indicated using the standard `409 Conflict` status code. To recover the client should start again from step 1. by retrieving the latest state of the resource. After recomputing the change locally, the client can once again attempt to update the resource.

As with most optimistic protocols, this solution works well if the ratio of updates (`PUT` or `POST`) to reads (`GET`) is small. The pattern should not be used for resources that are hotly contested between multiple clients, or in case the cost of re-trying a failed update is expensive.

5 Technologies

Over the past few years, REST has evolved from the original state in which an apparent lack of tooling support was limiting the adoption of the technology [27] and there have been quite a few frameworks that have been proposed for most programming languages and service delivery platforms (Table 1). Indeed, as a reaction to the complexity of WS-* technology stacks, REST was initially positioned as a lightweight alternative where no tools beyond a Web browser and some standard HTTP library were necessary to develop RESTful Web services. The situation has changed and with the growth in popularity of REST also a number of development frameworks have appeared.

5.1 Frameworks

Most frameworks support both client-side consumption of resources as well as server-side publishing of resources. However, some frameworks are starting to appear which specifically target the development of loosely coupled clients (e.g., RESTAgent or Guzzle). Some frameworks (e.g., ActiveResource, Compojure-rest, the Django REST Framework) are built as an extension of existing Web/MVC application development frameworks. Others (e.g., Persevere) come with a standalone HTTP server stack. Concerning the Java language, the oldest framework is RESTlet, while others (e.g., Jersey, RESTEasy, ApacheCXF) implement the JSR-311 [15] standard, which defines how to publish Java code as a RESTful Web service using source code annotations. With the 3.5 release of the .NET framework, also the Windows Communication Framework (WCF) technology stack supports REST. Likewise many existing WS-* technology frameworks (e.g., ApacheCXF) have begun to offer SOAP-less bindings to plain HTTP and started to support the use of JSON inside HTTP payloads.

5.2 Guidelines for framework selection

In general, it currently remains challenging to find a suitable framework which gives simple and correct guidance [32] to the service developer according to the REST constraints and which at the same time gives full access and control over the raw HTTP interactions. Even if it is possible to reuse or extend existing Web application development frameworks based on the Model-View-Controller (MVC) pattern, these may only offer limited support for processing both incoming and outgoing representations in customized non-HTML media types. Like in [26], we collect and discuss here a set of basic features that

Framework	Language/Platform	Project Homepage
ActiveResource	Ruby/Rails	http://api.rubyonrails.org/classes/ActiveResource/Base.html
apache2rest	PERL	http://code.google.com/p/apache2rest/
ApacheCXF	Java	http://cxf.apache.org/
Bowler	Scala	http://bowlerframework.org/
C2Serve	C++	http://www.c2serve.eu/
Compojure-rest	Clojure	http://github.com/ordnungswidrig/compojure-rest
Crochet	Scala	https://github.com/xllora/Crochet
Django REST	Python/Django	http://django-rest-framework.org/
Exyus	.NET	http://code.google.com/p/exyus/
FRAPI	PHP/Zend	http://getfrapi.com/
Guzzle	PHP	http://guzzlephp.org/
Jersey	Java	http://jersey.java.net/
OpenRASTA	.NET	https://github.com/openrasta/openrasta/wiki
Persevere	JavaScript	http://www.persvr.org/
Pinky	Scala	https://github.com/pk11/pinky/wiki
Piston	Python/Django	https://bitbucket.org/jespern/django-piston/wiki/Home
prestans	Python/WSGI	http://prestans.googlecode.com/
Recess	PHP	http://www.recessframework.org/
RESTAgent	Java	http://restagent.codeplex.com/
RESTEasy	Java	http://www.jboss.org/resteasy.html
RESTfulie	Ruby, Java, C	http://restfulie.caelum.com.br/
RESTify	JavaScript/Node	http://mcavage.github.com/node-restify/
RESTlet	Java	http://www.restlet.org/
RESTSharp	.NET	http://restsharp.org/
Scotty	Haskell	https://github.com/xich/scotty
Spray	Scala/Akka	http://spray.cc/
Taimen	Java, Clojure	https://bitbucket.org/kumarshantanu/taimen/
Tonic	PHP	http://peej.github.com/tonic/
Webmachine	Erlang	http://wiki.basho.com/Webmachine.html
Yesod	Haskell	http://www.yesodweb.com/
WCF	.NET	http://msdn.microsoft.com/en-us/library/vstudio/bb412169.aspx
WebPy	Python	http://webpy.org/
Wink	Java	http://incubator.apache.org/wink/

Table 1 Technology: Frameworks for developing and hosting RESTful Web Services (Homepage links verified as of 1^{st} October 2012)

should be supported by a fully featured framework for developing RESTful Web services.

- Can requests be routed to the corresponding service logic based on both resource identifiers and HTTP methods? Some frameworks only use resource identifiers and ignore methods, leaving it up to the developer to run different logic based on the request method.
- Are custom or extended HTTP verbs supported? Can the framework support different URI schemes or is it tied to HTTP/HTTPS URIs? Even if

REST does not make any assumption about the actual uniform interface, most frameworks are tightly couled with the HTTP protocol and thus assume that only the HTTP methods will be used.
- Does the framework enforce the semantics of the HTTP uniform interface (i.e., read-only GET, idempotent PUT and DELETE)?
- What is the abstraction level required to handle content type negotiation? Can the same service logic be easily reused for responses returned using different representation formats? Is the developer required to manually work with HTTP headers? Can custom media types be defined?
- Are ETags headers automatically computed and checked? How does the framework deal with conflicting updates?
- What are the assumptions made by the framework concerning the lifecycle of a resource? Can different business logic be invoked depending on the state of a resource? Is the state of a resource persisted implicitly or explicitly across server reboots?
- How easy is it to embed links to related resources in a representation being sent back to the client?
- Are URI templates supported for request routing and link generation? Must URI templates be embedded in the source code, or can they be read from configuration files, or can they be dynamically discovered and remotely updated?
- Does the framework transparently handles redirects to new resource identifiers?
- How easy is it to configure caching support without rebuilding the service logic and without relying on external caching proxies?
- How does the framework map internal exceptions of the service logic to HTTP status codes? Can such mapping be customized?
- Does the framework present REST as an optional "transport protocol binding" next to WS-* technology, or is REST the default, or the only option?
- How difficult is to configure the framework to use HTTPS?
- Does the framework support some notion of service interface description? Can such description be generated automatically for documentation purposes? Can code be generated from the description?
- Does the framework allow to automatically scale-out the service on multiple parallel processing units in a multicore or a cluster environment? How does the framework deal with concurrency?

These questions should be considered when evaluating the adoption of one of the currently emerging frameworks for supporting the development and operation of RESTful Web service APIs. Due to space limitations and given the current state of flux of the technology, we have chosen not to include any assessment on how the various frameworks listed in Table 1 would comply with the features mentioned in the previous checklist. A very good survey addressing a subset of the features and of the frameworks has recently appeared in [32].

6 Discussion

Service-oriented architectures promote the design of distributed and integrated systems out of the composition of reusable and autonomous services [18, 9]. The goal is not only to reduce integration costs through the standardization of interface contracts and the interoperability of middleware tools [2], but also to lower the efforts needed to manage the evolution of the integrated systems thanks to the loose coupling that is established among its constituent services. The design constraints of the REST architectural style help to achieve such quality attributes not only in the context of the Web but also when applied to the design of Web service APIs. In particular, reuse [29] and loose coupling [20] are emphasized by employing a uniform interface for all elements within the same architecture; performance and scalability are supported by ensuring the visibility of the interactions, which are kept stateless, and introducing intermediary caching layers where appropriate; interoperability is fostered by the wide-reaching standardization of the underlying technologies (i.e., HTTP, URIs, SSL) as well as the opportunity for dynamic negotiation of the most understandable representation format; reliability is enhanced through the use of idempotent interactions, which can be automatically retried in case of failures.

In the context of service oriented architectures, REST promotes the use of a novel (or different) kind of software connector to coordinate the interactions between a set of distributed services. As opposed to traditional bus connectors for services which enable to use primitives such as synchronous remote procedure calls (RPC) or asynchronous messaging (à la publish/subscribe), REST resources enable the reliable transfer and sharing of state between multiple services. As illustrated in this chapter's example, the state of a poll resource can be shared by multiple participants by means of its resource identifier. By initializing a new poll, one client can post information – literally on the Web – with the intention of sharing this information with other clients, which can then manipulate it to find an agreement. Whereas each interaction between the client and the resource makes use of synchronous HTTP request/response rounds, the overall end-to-end interaction between multiple clients mediated by the resource is completely asynchronous. As long as the various clients can discover the identifier of the shared resource, they can exchange information through it without ever being directly in contact with one another. To this extent, REST introduces a different interaction style between services that is more similar to the one enabled by a blackboard or a tuple-space software connector, rather than a messaging and publish/subscribe system used in most traditional service-oriented architectures.

7 Conclusion

The Web can be seen as an existence proof that it is known how to build highly scalable, decentralized and loosely coupled distributed systems. The architectural principles explaining how the Web works can thus be adopted to build integrated, service-oriented systems that could also be expected to feature similar quality attributes. This is the claim of RESTful Web services, which advocate the correct and complete use of the HTTP protocol for the design and the delivery of Web service APIs. Over time a number of patterns have appeared to complement the basic guidance found within the original design constraints of the REST architectural style. These patterns describe conventional solutions for specific design problems within the context of the existing standard HTTP protocol. From the technology perspective, a clear need for supporting the automated development and hosting of RESTful Web services has been addressed by the growing number of emerging frameworks with variable degrees of stability and maturity.

8 More information

In addition to the original formulation of REST in Dr. Fielding's dissertation [11], more information about REST and RESTful Web services can be found in several books that have been published on the subject in the past few years. [24] introduces the term RESTful Web services; [1] provides a collection of best practices explaining how to make correct usage of the HTTP protocol; [30] gives an extensive and well written discussion on how to use the Web as an integration platform; [25] has a similar goal but targets the German-speaking audience; [31] is a collection of research-oriented, application-oriented and practice-oriented writings on REST; [3] promotes the term Hypermedia API, focusing on the least known aspects of REST. [10] gives an in-depth discussion of the relationship between SOA and REST. More design patterns for RESTful Web services can be found in [7].

Acknowledgements Many thanks to Erik Wilde for all the help in preparing and running four editions of a successful WWW and ICWE tutorial on RESTful Web Services.

References

1. Allamaraju, S.: RESTful Web Services Cookbook. O'Reilly & Associates, Sebastopol, California (2010)
2. Alonso, G.: Myths around web services. Bulletin of the Technical Committee on Data Engineering **25**(4), 3–9 (2002)
3. Amundsen, M.: Building Hypermedia APIs with HTML5 and Node. O'Reilly (2011)
4. Berners-Lee, T.: Long live the web. Scientific American (12) (2010)
5. Berners-Lee, T., Cailliau, R., Luotonen, A., Frystyk Nielsen, H., Secret, A.: The world wide web. Communications of the ACM **37**(8), 76–82 (1994). DOI 10.1145/179606.179671
6. Berners-Lee, T., Fischetti, M., Dertouzos, M.: Weaving the Web. HarperCollins, San Francisco, California (1999)
7. Daigneau, R.: Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison Wesley (2011)
8. Dusseault, L., Snell, J.M.: Patch method for http. Internet RFC 5789 (2010)
9. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall (2005)
10. Erl, T., Carlyle, B., Pautasso, C., Balasubramanian, R.: SOA with REST: Principles, Patterns and Constraints for Building Enterprise Solutions with REST. Prentice Hall (2012)
11. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine, Irvine, California (2000)
12. Fielding, R.T., Gettys, J., Mogul, J.C., Frystyk Nielsen, H., Masinter, L., Leach, P.J., Berners-Lee, T.: Hypertext transfer protocol — http/1.1. Internet RFC 2616 (1999)
13. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Transactions on Internet Technology **2**(2), 115–150 (2002). DOI 10.1145/337180.337228
14. Goland, Y.Y., Whitehead, E.J., Faizi, A., Carter, S., Jensen, D.: Http extensions for distributed authoring — webdav. Internet RFC 2518 (1999)
15. Hadley, M., Sandoz, P.: Jax-rs: The java api for restful web services. Java Specification Request (JSR) 311 (2009)
16. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating web apis on the world wide web. In: Proc. of the 8th IEEE European Conference on Web Services (ECOWS2010), pp. 107–114 (2010). DOI 10.1109/ECOWS.2010.9
17. Nielsen, J.: User interface directions for the web. Communications of the ACM **42**(1), 65–72 (1999). DOI 10.1145/291469.291470
18. Papazoglou, M.P., van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. VLDB Journal **16**, 389–415 (2007)
19. Parastatidis, S., Webber, J., Silveira, G., Robinson, I.: The role of hypermedia in distributed system development. In: Pautasso et al. [21], pp. 16–22. DOI 10.1145/1798354.1798379
20. Pautasso, C., Wilde, E.: Why is the web loosely coupled? a multi-faceted metric for service design. In: J. Quemada, G. León, Y.S. Maarek, W. Nejdl (eds.) 18th International World Wide Web Conference, pp. 911–920. ACM Press, Madrid, Spain (2009)
21. Pautasso, C., Wilde, E., Marinos, A. (eds.): First International Workshop on RESTful Design (WS-REST 2010). Raleigh, North Carolina (2010)
22. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. "big" web services: Making the right architectural decision. In: J. Huai, R. Chen, H.W. Hon, Y. Liu, W.Y. Ma, A. Tomkins, X. Zhang (eds.) 17th International World Wide Web Conference, pp. 805–814. ACM Press, Beijing, China (2008)
23. Richardson, L.: Developers like hypermedia, but they don't like web browsers. In: Pautasso et al. [21], pp. 4–9. DOI 10.1145/1798354.1798377

24. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly & Associates, Sebastopol, California (2007)
25. Tilkov, S.: REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien. dpunkt.verlag, Heidelberg, Germany (2009)
26. Tilkov, S.: REST litmus test for web frameworks (2010). `http://www.innoq.com/blog/st/2010/07/rest_litmus_test_for_web_frame.html`
27. Vinoski, S.: Restful web services development checklist. IEEE Internet Computing **12**(6), 94–96 (2008). DOI 10.1109/MIC.2008.130
28. Vinoski, S.: Rpc and rest: Dilemma, disruption, and displacement. IEEE Internet Computing **12**(5), 92–95 (2008)
29. Vinoski, S.: Serendipitous reuse. IEEE Internet Computing **12**(1), 84–87 (2008). DOI 10.1109/MIC.2008.20
30. Webber, J., Parastatidis, S., Robinson, I.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly & Associates, Sebastopol, California (2010)
31. Wilde, E., Pautasso, C. (eds.): REST: From Research to Practice. Springer-Verlag, Heidelberg, Germany (2011)
32. Zuzak, I., Schreier, S.: Arrested development: Guidelines for designing REST frameworks. Internet Computing **16**(4), 26–35 (2012)