Designing Web-Based Mobile Services with REST

Claudio Riva and Markku Laitkorpi

Nokia Research Center, P.O. Box 408,
FIN-00045 NOKIA GROUP, Finland
{claudio.riva,markku.laitkorpi}@nokia.com

Abstract. The Web is emerging as the favorite platform for delivering applications and services. The REST architectural style comprises the key principles behind its design and success. While REST is originally defined in the context of publishing hypermedia documents, it is becoming a popular method for implementing Web services as well. The goal of this paper is to explore the principles of the REST style for building mobile services and to address the mobile specific constraints. We present the design method that we have followed for building a basic photo storage service. Our preliminary evaluation confirms that REST is a flexible and extensible approach for building mobile services.

Keywords: Software architectures, mobile services, service oriented architectures, web oriented architectures, REST.

1 Introduction

New compelling mobile services are expanding the capabilities of mobile devices besides the traditional services of voice and short messaging. Nowadays modern mobile devices have built-in support for several mobile services like web browsing, mobile email, mobile search, photo sharing, news readers, internet radio and navigation. The promise of the mobile Internet is turning into reality for many consumers. In some cases, the mobile services are tightly bundled with the devices in order to maximize their user experience, e.g. Blackberry Research in Motion's (RIM) push email service, Nokia N-Gage Arena and Nokia S60's support for mobile search and photo sharing with Flickr. With the availability of fast and convenient mobile Internet, we expect that mobile services will represent a significant value-adding element of future devices.

The World Wide Web is emerging as the favorite platform for delivering applications and services. The Web 2.0 phenomenon is exploiting the web browser as a cross-platform runtime environment for shipping and executing applications. Specialized libraries (e.g. Ajax) and frameworks (e.g. Ruby on Rails) have been developed for facilitating the creation of interactive web applications. The Web is also becoming the preferred platform for delivering services to third parties. Internet companies open their functionality and their data to the external developers through public interfaces implemented with Web services. These public interfaces, typically called Open APIs or Web APIs, are key enablers of the Web 2.0 phenomenon.

D. Kaeli and K. Sachs (Eds.): SPEC Benchmark Workshop 2009, LNCS 5419, pp. 439–450, 2009.

There are two dominant paradigms for creating Web APIs, namely SOAP/WS-* and REST. SOAP is an XML based protocol for exchanging messages over a computer network, typically tunneling over HTTP, and is the foundation for the more complex Web Services stack, also known as WS-* [10,11]. SOAP provides different types of messaging patterns, but the remote procedure call is commonly used when building Open APIs. A SOAP based Open API typically provides a set of remote operations that can be invoked over the Web.

REST (Representation State Transfer) follows a different philosophy than SOAP by focusing on data instead of operations. Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) [2], coined the term REST[1] in his doctoral dissertation [1]. REST is an architectural style derived from the Web, and its architectural elements and constraints aim at collecting the fundamental design principles that enable the great scalability, growth and success of the Web. REST is originally defined in the context of publishing hypermedia documents but it has become a popular method for publishing Web services as a Web-friendly alternative to SOAP. REST is primarily focused on defining and addressing web resources (like documents and images) and for managing their representations. Although there is only one version of REST defined, it does not mean that every system is supposed to be equally RESTful. Therefore, the constraints of REST should be taken as a toolkit that helps the architect identify the benefits or costs, depending on whether the actual requirements fit the constraints.

The goal of the present research is to investigate how to apply the principles of the RESTful design to the development of mobile services. In particular, our focus is to analyze the benefits and limitations of REST in a mobile context and to elaborate a proper design process. Designing REST services is about modeling resources that are published on the Web and the functionality is implemented with a limited set of operations on those resources. The design process resembles the data-driven design processes (e.g. for databases). In this paper, we present our preliminary work of designing REST services. First, we present the key rationale and constrains of REST design in a mobile environment. Then, we present our design process and we apply it to the case study of designing a photo storage services. Finally, we conclude with a preliminary evaluation.

2 Mobile Services with REST: Rationale and Constraints

Mobile services allow users to access network services through the user interface of a mobile device. They consist of a *client application*, the client component that provides the user interface on the mobile device, and a *service backend,* the server component in the network that provides the remote functionality.

The implementation of the client application heavily depends on the capabilities of the mobile device and the available run-time environments. Often, the mobile software developer must specialize the client applications to the various clients in order to take full advantage of the device specific features [5]. The client application also

[1] There are different interpretations of REST. Unless otherwise specified in this paper we refer to a truly resource-oriented approach, often referred as RESTful.

needs to efficiently interact with the backend in order to fetch the data that is presented in the user interface. There are several motivations to design the backend according to the REST style:

- It provides a uniform method for accessing the resources independently of the run-time environment (native, Java, Python, etc). Any client that supports the HTTP protocol is capable of accessing the REST services without the need of any additional messaging framework.
- The representation of a resource can be negotiated at run-time between the client and the server. This enables us to support mobile optimized variants of the resources according to the needs of the device.
- The REST principles favor scalability and fast response times (e.g. through caching). This is an important point for mobile services that potentially have a huge user base of billions of users. Scalability is often a hard to reach quality if not properly addressed early in the design.

However, the mobile environment poses additional challenges and constraints over the typical REST design. We summarize them below:

- Network latency: mobile data networks (2G, EDGE, 3G) are optimized for content download. The bandwidth is often sufficient for most of the scenarios (like video streaming), but the network latency is still a major issue. Even if future networks beyond 3G (e.g. HDSPA, WiMAX) will minimize this problem, a reliable mobile service should make few assumptions about the underlying data network. Optimizing for high latency network is a key requirement for success. The roundtrip time for sending a request and receiving a response can be as high as two seconds. To avoid performance bottlenecks, the number of message exchanges with the server must be kept at the minimum. This requirement is often conflicting with REST services developed for low latency networks.
- Data formats: verbose data formats (e.g. XML) are slow to transfer and to parse. Mobile devices require lightweight formats. Compact binary formats are often the optimal solution but they compromise the service interoperability. Often, the tradeoff is between a proprietary optimization and an interoperable standard.
- Caching policy: to reduce the network traffic the mobile services must cache the data on the device memory. Caching can happen either at the protocol level (e.g. the HTTP responses) or at the application level (e.g. the actual data).
- Offline/online behavior: irregular network connectivity is typical for mobile devices. Mobile services must define a strategy for supporting offline operations.
- Thin vs. thick clients: how much logic resides on the client and on the server.

3 Designing REST Services

The core entities of the REST design are the web resources. To design the APIs we follow a data-driven design process that starts by modeling the domain of the APIs. In the following sections we describe our basic design principles, the design process and the structure of the REST API.

3.1 Design Principles

The basic architectural style is REST with the HTTP. The functionality is defined in terms of resources that are manipulated via HTTP as an application protocol.

A resource is an entity that has:

- name and address identified by a URI [3]
- one state at a time defined by a set of attributes and their values
- at least one representation that encode the current state in a particular content type

The allowed actions on the resources are based on the HTTP protocol with the following semantics:

- `GET`: retrieves a representation of a resource. This operation is safe with no changes that a client could be held accountable for.
- `PUT`: rewrites a resource with a new state that is enclosed in the request. This operation is idempotent (request can be re-processed with the same final state).
- `POST`: appends the entity enclosed in the request to the resource. A new resource might be created. This action returns either the location of the created resource or the resource representation after appending. This operation is not idempotent.
- `DELETE`: removes the mapping between a resource and its URI, thus making the resource inaccessible. This operation is idempotent.

Each request is authenticated independently using the standard HTTP authorization header. Hence, there is no need for session control and cookies. The access control for a resource is conducted at the URI level.

Concerning the data objects, we support the exchange of two types of structured data: containers (e.g. a collection of photos) and single items (e.g. an image). Besides the standard content types for media (MIME types [6]), we support content types for structured data for different applications:

- ATOM (`application/atom+xml`): widely popular XML based format for blogs and mash-ups [7,8]
- JSON (`application/json`): lightweight format for structured data [9]
- binary JSON (`application/x-bjson`): our own extension to JSON format to support the exchange of binary data.

We also define a decoration mechanism for clients to control the exact content of the message body of the HTTP responses. In the header of the HTTP request, we can add a custom header called `x-decorations` that contain a comma separated list of decorations to be applied to the response. We have defined the following decorations:

- `xlink-id`: add basic identifier links to objects
- `xlink`: adds all available links to objects
- `item(id | summary | full)`: controls the verbosity of the items when listed in a container
- `inline(<content-type>)=<link name>`: outputs an optimized response that includes the associated data in one single request-response cycle. The output

format depends on the content type of the resource. For JSON and XML we use `multipart/related`, for binary JSON and ATOM we inline the data in place. The `<content-type>` can be any of the allowed content types of the linked resource (e.g. `image/jpg`). The `<link-name>` can be any of the link names published by the resource (e.g. `tn_tiny`)

3.2 Design Process

Designing REST services involve modeling web resources and their URIs. Therefore, we follow a data-driven approach that is summarized in three steps:

1. Developing the abstract data model. The first step is to identify the concepts and their associations that are part of the API. For this task, we follow the traditional techniques of object-oriented analysis and design, focusing on all the relevant nouns in the problem domain.
2. Deriving the core resource model. Based on the abstract data model, we identify the *primary resources* of the REST design, i.e. the central concepts of the API. We distinguish between two types of resources: *containers* (collections of items) and *items*. Besides the primary resources, we can also identify the *secondary resources* that can be accessed from the primary resources through their associations.
3. Designing the URI space of the API. The final step is to define the URI space of the API. For each resource we define the URI, allowed methods, content types, and links to other resources. The structure of the API is described in the next section.

3.3 API Structure

The structure of the API is documented using four elements that are described below:

– URI space. Each resource (either a container or an item) is addressed by one URI. The name containment implies strong sub-resource relationship. We use the following syntax:

`../<ownership>/<primary projection>/<association>?<query>`

where:

`<ownership>`: encodes the owner of the resource, such as user “john”
`<primary projection>`: encodes the primary resource, such as “photos” or “photo”
`<association>`: encodes the associations from the primary resources, such as “tags of a photo”
`<query>`: query string for multidimensional projections.

– Methods and content types: this describes the available interactions on the resources and the supported content types
– Links: traversable connections from one resource to another. It is a combination of a name (semantics) and a URI (concrete address)
– Representations: actual data content of the resource

4 Example of REST API for Photos

As a case study we have designed the potential REST API for a photo storage service. The service should provide the functionality for uploading new photos, tagging them, organizing them into folders and assigning properties. There are many similar services on the Web that provide an API for storing photos: Flickr[2] and Smugmug[3] as examples, but they are clearly operation-centric and only accidentally RESTful. The goal of our case study is to design a photo storage service that follows the REST principles and is usable from mobile devices. We describe the data model, the list of web resource and several use cases to access the service.

4.1 Data Model

The Figure 1 shows the simple data model for the photo storage service and their attributes. We have included the following concepts:

- **User:** the authenticated user of the system. Each user has a unique *login name*
- **Photo:** the data object for a photo. Each photo has a unique id.
- **Bag:** a user-specific label for grouping photo objects in albums
- **Tag:** a publicly defined label that can be associated to a photo object
- **Content:** the actual binary representation of the photo (e.g. the jpg file)
- **Attr:** an attribute that assigns a value to a property
- **Prop:** a property definition with semantics, defined by a scope and a name.
- **Tn:** the binary representation of a thumbnail of a photo.

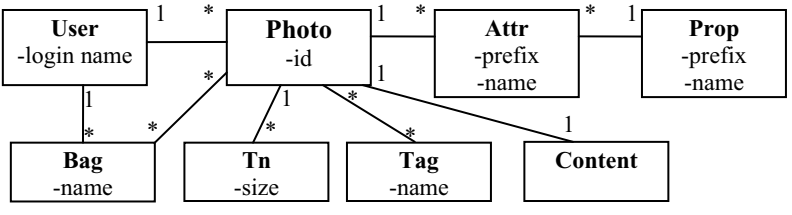

Fig. 1. Abstract data model of the Photo API

4.2 Core Resources

From the data model we can directly derive the core resources of the REST API. Below we list the core resources:

Primary resources

Containers:	Photos, Tags, Bags, Properties
Items:	Photo, Tag, Bag, Property

Secondary Resources

Containers:	Thumbnails of a photo, Attributes of a photo, Tags of a photo, Bags of a photo
Items:	Content, Thumbnail, Attribute

[2] http://www.flickr.com/services/api
[3] http://www.smugmug.com

4.3 API Structure

Based on the core resource model, we define the structure of the URIs. Table 1 lists the URIs of the resources and Table 2 documents the semantic of the HTTP actions.

Table 1. Structure of the API for Photos

URI			Description
Ownership	Primary projection	Association	
Photos			
/users/{*name*}	/photos		Photos for user {*name*} (container)
/users/{*name*}	/photos/recent		Recent photos for {*name*} (container)
/users/{*name*}	/photos/{*photo-id*}		Photo item
/users/{*name*}	/photos/{*photo-id*}	/content	Photo content of {*photo-id*}
/users/{*name*}	/photos/{*photo-id*}	/tns/tiny	Photo thumbnail (50x50)
/users/{*name*}	/photos/{*photo-id*}	/tns/small	Photo thumbnail (100x100)
/users/{*name*}	/photos/{*photo-id*}	/tns/medium	Photo thumbnail (200x200)
/users/{*name*}	/photos/{*photo-id*}	/tns/large	Photo thumbnail (400x400)
/users/{*name*}	/photos/{*photo-id*}	/tns/custom/{*xsize*};{*ysize*}	Photo thumbnail xsize x ysize
/users/{*name*}	/photos/{*photo-id*}	/attrs	Photo attributes for {*photo-id*} (container)
/users/{*name*}	/photos/{*photo-id*}	/attrs/{*attr-prefix*}/{*attr-name*}	Photo attribute
/users/{*name*}	/photos/{*photo-id*}	/tags	Tags for {*photo-id*} (container)
/users/{*name*}	/photos/{*photo-id*}	/tags/{*tag-name*}	Photo tag
/users/{*name*}	/photos/{*photo-id*}	/bags	Bags for {*photo-id*} (container)
/users/{*name*}	/photos/{*photo-id*}	/bags/{*bag-name*}	Photo bag
Tags			
	/tags		All tags (container)
	/tags/{*tag-name*}		Tag item
	/tags/{*tag-name*}	/photos	All photos with tag {*tag-name*} (container)
Bags			
/users/{*name*}	/bags		All bags for user {*name*} (container)
/users/{*name*}	/bags/{*bag-name*}		Bag item
/users/{*name*}	/bags/{*bag-name*}	/photos	All photos in bag {*bag-name*} (container)
Properties			
	/props		Public properties (container)
	/props/{*prop-prefix*}/{*prop-name*}		Public property

Table 2. Supported methods and content-types for the resources. Unless otherwise specified all the methods support the following content-types: `application/json,application/atom+xml, application/x-bjson`.

Resource	Method	Content-type	Description
Photos (container)	GET		Retrieves the array of photos
	POST	image/jpeg	Appends a new photo
Recent photos (container)	GET		Retrieves the array of recent photos
Photo item	GET		Retrieves the photo item
	PUT		Replaces the content of the item
	DELETE		Removes the item
Photo content	GET	image/jpeg	Retrieves the image file
	PUT	image/jpeg	Replaces the image file
Photo thumbnail	GET	image/jpeg	Retrieves the thumbnail file
Photo Attributes (container)	GET		Retrieves the array of attributes
	POST		Adds a new attribute to a photo
Photo Attribute	GET		Retrieves the value of an attribute
	PUT		Updates the value of an attribute
Tags (container)	GET		Retrieves the array of tags
	POST		Add a new tag to a tag container
Tag item	GET		Retrieves the tag data
	PUT		Updates the tag data
	DELETE		Removes the tag
Bags (container)	GET		Retrieves the array of bags
	POST		Adds a new bag to a bag container
Bag item	GET		Retrieves the bag data
	PUT		Replaces the bag data
	DELETE		Removes the bag (but not the content)
Properties (container)	GET		Retrieves the array of properties
	POST		Creates a new property
Property item	GET		Retrieves the data about a property
	PUT		Updates a property
	DELETE		Removes a property

4.4 Use Cases How to Access the Photo API

Uploading a photo
The user can upload a photo by sending a `POST` request to a Photo container (e.g. `/users/mary/photos`) the binary data that contains the jpg image of the photo. If successful, the services replies with the message "`201 Created`" and the Location header contains the link to the created resource.

Fetching the list of photos
The user can fetch the list of photos stored in the system with a `GET` request on the root Photo container (e.g. `/users/mary/photos`). The response in JSON format is an array with the IDs of the photos:

```
{"photos": {"photo": [{"id": 169}, {"id": 170}, {"id": 171},
{"id": 172}, {"id": 173}, {"id": 174}, {"id": 175}, {"id":
176}], "total": 8, "start": 0, "count": 20}}
```

By passing the inline header in the request the user can request additional information to be added in the response. For instance, we can inline the thumbnails by adding the header: `x-decorations: inline(image/jpeg)=tn_tiny`. The response is a multipart message containing the JSON data and the thumbnails.

Fetching the content/thumbnail of one photo
The user can fetch the content of one photo with a `GET` request on the Photo resource (e.g. `/users/mary/photos/169/content`). The response is the binary data of the jpeg image. The thumbnail can be retrieved from the suitable thumbnail URI (e.g. `/users/mary/photos/169/tns/small`).

Tagging photos
Because a single tag is simply like a label, the user can add a new tag to a photo by sending a `PUT` request to a non-existent Tag resource of the Photo (e.g. `/users/mary/photos/169/tags/Holiday`). Alternatively, the user can upload multiple tags by using a POST request on the Tags container resource with the body containing the tag names in JSON format (e.g. `{"tags": {"tag": [{"name": "Holiday"}, {"name": "Maledives"}]}}`)

The user can fetch the list of all the photos with a particular tag by sending a `GET` request to the Tag resource (e.g. `/tags/Holiday/photos`). The result is an array of photos:

```
{"photos": {"photo": [{"id": 169}, {"id": 170}], "total": 2,
"start": 0, "count": 20}}
```

The user can un-tag a photo sending a `DELETE` request to the Tag resource of the Photo (e.g. `/users/mary/photos/169/tags/Holiday`).

Attaching the geo location attribute to one photo
The user can define new properties for the Photo resource and attach the attributes to a particular Photo. In this case we need to define a new property for storing the geo location of a photo (e.g. the property `/loc/coord`). First, we send a `POST` request to the Properties container (`/props`) with the new property to create: `{"prop": [{"name": "coord", "prefix": "loc"}]}`. Then, we can add the attribute to the photo by sending a `PUT` request to the location of the photo (e.g. `/users/mary/photos/169/loc/coord`) with the following body: `{"attr": {"value": "45.2 10.7"}}`.

5 Preliminary Evaluation

5.1 Prototype Implementation

We have prototyped a reference implementation of the photo service that was presented in the previous section. The implementation is based on the Ruby on Rails

framework [4] that provides several convenient features for the implementation of REST services. To evaluate the benefits and the limitations of the REST design we have tested the backed with the three different clients:

- **Maemo.** It is the linux-based platform for the Nokia internet tables 770 and N800. We implemented the client using the Python scripting language.
- **Java MIDP.** The Java MIDP environment runs on most mobile devices. Java MIDP supports the HTTP protocol but not all the HTTP actions (only GET and POST are supported). We have implemented a simple workaround for emulating the PUT and DELETE operations with POST. We tested the client on a Nokia N73 device.
- **Python for S60.** S60 is Nokia's platform for smartphones and multimedia computers. We have implemented the client using the port of the Python language on S60. The port provides full support for the standard HTTPlib module.
- **Mash-ups.** We have assembled our photo service with the services from other Internet sites (e.g. Google maps) within the browser environment. The mash-ups have been implemented in Javascript.

5.2 Main Observations

Our general evaluation of REST for mobile services is positive but there are still several limitations that need to be solved. We list the main points of our evaluation below:

- REST with HTTP provides a uniform mechanism for building web services that requires a minimal infrastructure on the client and the server sides. Most importantly, this approach is independent of the run-time environment, as long as there is decent support for HTTP as a common denominator. Unfortunately, some run-time libraries targeting mobile devices still treat HTTP as a simple transport mechanism instead of a proper application protocol. For example, MIME multipart support is typically provided by the mail libraries, and its API is often incompatible with HTTP message processing API, thus rendering HTTP multipart message parsing rather clumsy and inefficient.
- The network latency is a concrete problem for web services, especially for the overall user experience on the client-side. The user interface requires a proper design to take into consideration the long roundtrip periods for transferring the data across the wireless network. We have experimented with two solutions that can be used for both read and write requests. The first one is to inline additional data in one request (e.g. inlining the thumbnails) that allows fetching or updating. However, this solution inhibits the individual HTTP caching of the inlined resources. The second one is to create convenient URIs that provide more application-specific data either by bypassing or augmenting the core API, effectively creating custom views on top of the underlying core resources. When properly designed in terms of resources and their URIs, these views can then be used for batch updates with a single request, for example. However, this approach complicates the design of the back-end, because it may introduce potentially complex application-specific aspects on the server side.

- The JSON data format suits very well the mobile environment. It is simple to generate, to parse, and compact to transfer. According to our preliminary measurements, however, the size of the transferred data is not significantly smaller than that of XML data, especially when retaining the field names for self-descriptiveness. We have also extended the JSON format with binary data to support the response inlining, because at the moment we have found the multipart messages too complex to be processed on the client side.
- Designing simple and intuitive URIs is a critical part of the REST design. We have sketched a simple data-oriented design process but we need a method for specifying the resources with a proper formalism. Especially developers without prior experience of REST would greatly benefit from method and tool support that help them map their requirements to RESTful resource design. There is also the open issue of enabling the component-like reuse of resource subtrees across different parts of the API (e.g. reusing the Tag resource between the Photo and Video resource).
- In our solution, we do not have any explicit support for offline processing. Our RESTful approach, however, inherently fits many offline scenarios, mainly because of statelessness. For example, idempotent requests can be queued on the client side quite easily for later delivery, but this requires that dynamically created resources should have client-provided URIs.

5.3 "Mobile REST"

As explained above, the roundtrip cost is probably the most significant individual challenge in network-based mobile applications. Therefore, applying REST efficiently in mobile application architectures may require some additional constraints that guide the architects to make more mobile-friendly decisions. These constraints would primarily need to tackle the dilemma of "just enough requests, just enough data". In very simple terms, "just enough requests" refers to mechanisms that give clients enough control to adjust the scope of the RESTful interactions, thus reducing the need for multiple requests. As an opposite driving force, "just enough data" refers to mechanisms that clients may use to adjust the volume of data in a single message. Elaborating these additional constraints will be part of our future work.

6 Conclusions

The growth and success of the Web mainly derive from a thorough application of basic design principles that are crystallized in the REST style. REST is also becoming a popular paradigm for publishing Web services (as an alternative to the WS-* stack). In the current paper we have investigated how to apply the REST principles to the design of mobile services. We have identified several issues (like network latency and data formats) that need particular attention when applying the REST concepts to the mobile environment. For demonstration purposes, we have prototyped a REST photo web service and tested it with different mobile clients. Our preliminary evaluation shows that REST with HTTP provides a uniform mechanism for building web services that requires a minimal infrastructure on the client-side and is independent of

the run-time environment. The service is extensible with different data representations and, hence, allows us to properly address the mobile specific needs. The critical part of the design process is to elaborate a clear URI structure for the API, allowing mobile clients to make just enough requests and process just enough data. We have followed a data-driven design process but we have been lacking a meticulous design formalism (e.g., a UML based approach that helps developers map their requirements to REST constraints). Our future work will be focused on improving the design process of REST services, especially for the mobile environment, and conduct a more thorough evaluation with additional prototypes.

References

1. Fielding, R.T.: Architectural styles and the design of network-based software architectures, PhD Thesis, University of California, Irvine (2000)
2. Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. Internet RFC 2616 (June 1999)
3. Berners-Lee, T., Fielding, R.T., Masinter, L.: Uniform Resource Identifiers (URI): Generic syntax. Internet RFC 2396 (August 1998)
4. Thomas, D., Hansson, D.H.: Agile Web Development with Rails, Pragmatic Bookshelf, 2nd edn. (December 14, 2006)
5. van Gurp, J., Karhinen, A., Bosch, J.: Mobile Service Oriented Architectures. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 1–15. Springer, Heidelberg (2006)
6. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions. Internet RFC 2046 (November 1996)
7. Nottingham, M., Sayre, R.: The Atom Syndication Format. Internet RFC 4287 (December 2005)
8. Greogiro, J., de hOra, B.: The Atom Publishing Protocol, Internet Draft version 17 (November 2007)
9. Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON), Internet RFC 4627 (July 2006)
10. Web Services Architecture, W3C Working Group Note (February 11, 2004), `http://www.w3.org/TR/ws-arch/`
11. SOAP, Version 1.2, W3C Recommendation (Second edn.) (April 27, 2007), `http://www.w3.org/TR/soap/`