Experiences in Building a RESTful Mixed Reality Web Service Platform

Petri Selonen, Petros Belimpasakis, and Yu You

Nokia Research Center, P.O. Box 1000, FI-33721 Tampere, Finland
{petri.selonen,petros.belimpasakis,yu.you}@nokia.com

Abstract. This paper reports the development of a RESTful Web service platform at Nokia Research Center for building Mixed Reality services. The platform serves geo-spatially oriented multimedia content, geo-data like street-view panoramas, building outlines, 3D objects and point cloud models. It further provides support for identity management and social networks, as well as for aggregating content from third party content repositories. The implemented system is evaluated on architecture qualities like support for evolution and mobile clients. The paper outlines our approach for developing RESTful Web services from requirements to an implemented service, and presents the experiences and insights gained during the platform development, including the benefits and challenges identified from adhering to the Resource Oriented Architecture style.

Keywords: Web Services, REST, Mixed Reality, Web Engineering.

1 Introduction

Mixed Reality (MR) refers to the merging of real and virtual worlds to produce new environments and visualizations where physical and digital objects co-exist and interact in real time, linked to each other [8]. As a concept, it encompasses both Augmented Reality (AR) and Augmented Virtuality (AV). As smart phones become more potent through graphical co-processors, cameras and a rich set of sensors like GPS, magnetometer and accelerometers, they turn out to be the perfect devices for AR/MR applications: interactive in real time and registering content in 3D [2] [6].

MR related research has been active for many years at Nokia Research Center (NRC). Most AR/MR applications developed at NRC (e.g. [5][9]) have focused on studying user interface and interaction, and therefore been either stand alone clients or clients linked to project specific service back-ends. The lack of having open Application Programming Interfaces (APIs) have kept these applications isolated, resulting in limited amount of content, prohibited sharing it with other similar systems, and not providing support for third party mash-ups. Some AR/MR prototypes have utilized content from Internet services like Flickr that have not been built for AR/MR applications, and therefore lack proper support for rich media types and geographical searching. In practice, each new AR/MR application has implemented another backend to host their content.

B. Benatallah et al. (Eds.): ICWE 2010, LNCS 6189, pp. 400–414, 2010.

This paper describes an effort to move from stand-alone service back-ends to a common Mixed Reality Web services platform, referred to as MRS-WS. An essential aim was to build a common backend for easily creating Mixed Reality services and applications for both mobile clients and Web browsers. Taking care much of the complexity at the backend infrastructure side, the creation of applications on top of this service platform would be faster, simpler and open also to 3rd parties.

As we essentially serve content, we have chosen REST [4] as the architectural style for the platform. Up to date, there does not exist a systematic method for proceeding from requirements to an implemented RESTful API, so we briefly outline the approach emerged during the development. In what follows, we present the platform architecture, explore its architectural qualities and discuss some of the experiences and insights gained during building the platform.

2 Requirements for the Service Platform

The MRS-WS platform is developed within the context of a larger NRC Mixed Reality Solutions program building a multitude of MR solutions, each integrating with the platform. At the time of writing, a first client solution has been deployed and is currently being maintained, and a second one is under development. In addition to the functional features required by the dedicated program clients, there are other more generic requirements to the platform architecture, outlined next.

2.1 Overview to Functional Requirements

The first of the two clients allows the user to explore photos at the location of the image spatially arranged with its original orientation and camera pose. Once a photo is taken using a mobile phone, the photo is uploaded to a content repository with metadata acquired from the GPS, digital compass (magnetometer) and accelerometers of the device. The user and his friends can later add and edit comments, tags and descriptions related to the photos. There are both a Web browser based client for exploring the user's content rendered on map and 3D space (Augmented Virtuality) and a mobile client that can show the photo on location, overlaid on top of a view-through camera image *in situ* (Augmented Reality). Screenshots of this solution, called Nokia Image Space[1], is shown in Fig. 1. To enable access to masses of pre-existing content, the platform supports aggregation of content and social relationships from 3rd party content repositories, starting with Flickr.

The second client fetches street-view panoramas, building outlines and points of interest based on a location, and visualizes annotations and related comments belonging to the user overlaid on top the panorama. The client enables the users to annotate particular buildings and other landmarks through touch and share these annotations with other users. The geo-data is post-processed from commercial Navteq content using advanced computer vision algorithms for e.g. recognizing building outlines from panoramic images.

[1] `http://betalabs.nokia.com/apps/nokia-image-space`

Fig. 1. Image Space Web and mobile AR clients

Generalizing from the above client requirements and general MR domain knowledge, we can identify high-level functional requirements for the platform as support for

- managing MR metadata—geographical location, orientation and accelerometer data —for storing and visualizing content in 3D space;
- storing content of different media types, such as photos, videos, audio and 3D objects, as well as non-multimedia like point clouds and micro-blogging entries;
- managing the typical social media extensions for every content item, including comments, tags and ratings;
- managing application specific metadata for content items, allowing clients with specific needs to store pieces of information without modifications at the platform side;
- managing user identities, social connections and access control lists; and
- creating, modifying, retrieving, searching and removing the above content.

In addition, there are requirements related to content hosted on external repositories:

- aggregating content and social connections from, and synchronizing with, third party repositories to provide a uniform API to all content available to the clients, making the clients agnostic about the interfaces and existence of external repositories; and
- serving commercial geo-data and content post-processed from it, including 360-degree street view panoramic images, POIs and building outlines.

The platform is supporting both aggregating content from other content providers and exposing it in a uniform manner, and enabling other services to link our platform to them. The clients are able to discover, browse, create, edit and publish geo-content useful for MR applications.

2.2 Overview to Non-functional Requirements

In addition to the functional requirements collected above, we consider a few of the most important non-functional requirements to the platform, identified as

- usability of the platform to ensure that its users—end developers—find it easy to use and to integrate with;
- modifiability and extensibility of the platform to ensure that new requirements and clients can be rapidly integrated;

- support for mobile devices—the main devices through which augmented reality is realized—translating to efficient bandwidth utilization and fine-grained content control for the clients;
- security and privacy to ensure user generated content is safely stored and transferred; and
- scalability and performance to make sure the system architecture will scale up to be used in global scale with potentially millions of users.

While all of the above qualities are important, we place special emphasis in realizing the three first ones. The main contribution expected from the platform is to provide an easy-to-use API for developers to build MR applications and Web mash-ups. In the case of mash-ups, external services should be able to link their unique content to the MRS-WS platform to be seamlessly visualized in MR view by our platform clients. As a simple example, consider a restaurant business "pushing" their menu offering to our platform which would then be visible in AR view to a potential customer passing by the restaurant and exploring the area via his mobile phone, as a "magic lens" to reality. The key in the quest for killer applications is proper support for open innovation.

3 Developing Resource Oriented Architectures

REST architecture style provides a set of constraints driving design decisions towards such properties as interoperability, evolvability and scalability. To this end, RESTful APIs should exhibit qualities like addressability, statelessness, connectedness and adhering to a uniform interface. The MRS-WS platform serves what essentially is content: the core value of the service is in storing, retrieving and managing interlinked, MR enabled content through a unified interface. Therefore, REST and Resource Oriented Architecture [10] were chosen as the architecture style for the platform. We have also previously explored using REST with mobile clients, and the lessons learned were taken into use in the platform building exercise.

However, so far there does not exist a commonly agreed, systematic and well-defined way to proceed from a set of requirements to an implemented RESTful API. The best known formulation of designing RESTful Web services has been presented by Richardson and Ruby [10] which can be summarized as finding resources and their relationships, selecting uniform operations for each resource, and defining data formats for them. This formulation is too abstract to be followed as a method and further, it does not facilitate communication between service architects and other stakeholders. In our previous work [7] we have explored how to devise a process for developing RESTful service APIs when the set of service requirements are not content oriented but arbitrary functional features. We took the learnings of this work and constructed a more lightweight and agile approach, suited for the Mixed Reality program and its a priori content oriented domain. In a way, the result is something resembling a lightweight architecture style for developing Resource Oriented Architectures, in a way a ROA meta-architecture.

3.1 Resource Model, Information Model and Implementation

In the heart of the approach is the concept of a *resource model*. A resource model, adapted from Laitkorpi et al [7], organizes the elements of a domain model to addressable entities that can then be mapped to elements of a RESTful API, service implementation and database schema, while still being compact and understandable by the domain experts. The resource model divides resources into *items*, which represent individual resources having a state that can be created, retrieved, modified and deleted, *containers* that can be used for retrieving collections of items and creating them, and *projections* that are filtered views to containers. Resources can have subresources and references to other resources.

A natural way to map the service requirements into a resource mode is to first collect them to a domain model called the *information model*—expressed for example as a UML class diagram—which is essentially a structural model describing the domain concepts, their properties and relationships, and annotated with information about required searching, filtering and other retrieval functionality. The concepts of the information model are mapped to the resource model as resources; depending on their relationship multiplicities, they become either items or containers containing items. Composition relationships form resource–subresource hierarchies while normal relationships become references and collections of references between resources. Attributes are used to generate resource representations with candidates for using standard MIME singled out when possible. Each search or filtering feature defined by an annotation—e.g. UML note—is represented by a projection container. There are obviously fine-grained details in the mapping, but they are left out as they are not relevant for the high-level architectural picture.

The concepts of a resource model, i.e. containers, items and projections, are then mapped to implementation level concepts in service specifications (e.g. WADL), database schema and source code. For example, containers can manifest themselves as tables in relational databases, items as rows in a table, item representation as columns, and subresources and references as links to other tables. Similarly, containers and items can be mapped to HTTP resource handlers in the target REST framework (say, Ruby on Rails, Django or Restlet).

3.2 MRS-WS Implementation Binding and Architecture

The concrete binding between the MRS-WS domain model, resource model and the implementation is done by mapping resource model concepts above to the concepts of the selected technology stack: Restlet, Hibernate, Java EE and MySQL. The overall approach is illustrated in Fig. 2. The top-left corner shows a simple information model fragment, mapped to a resource model shown at bottom-left corner and further to an implementation shown on the right-hand side. Based on the information model fragment, the resource model contains Annotations container, Annotation item and Annotations-By-Category projection that are in turn mapped into implementation

model elements: AnnotationsResource, AnnotationResource, AnnotationsView, AnnotationView, Annotation model and AnnotationDAO. The binding is further explained in Table 1.

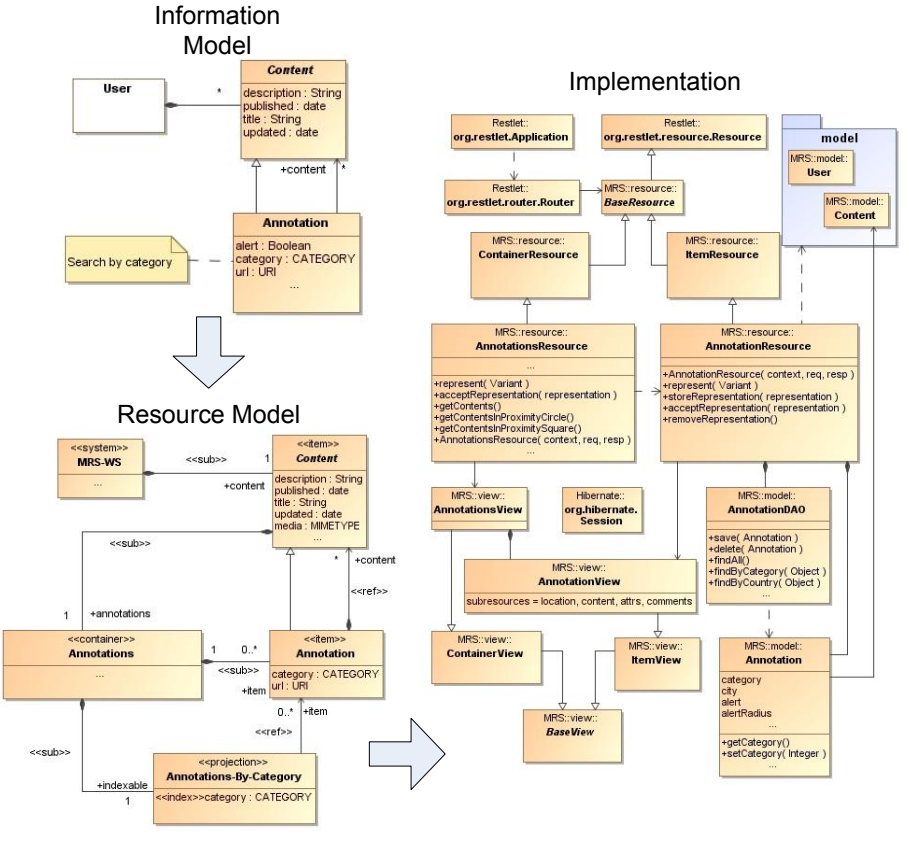

Fig. 2. Overview of the ROA architecture

Fig. 3. shows the architectural layers of the platform. The design of the architecture is to provide a uniformed REST API set for MR data plus mashed-up contents from third party content providers available to the service clients. Each resource handler is identified by a URI and represents the unique interface for one function and dataset. The operation function delegates the request to concrete business logic code which works closely in the persistence layer. After returning from the business logic code, the operation function transforms the returned internal data to the corresponding data format through the representation classes along the output pipe to the web tier and the client eventually.

Table 1. MRS-WS implementation binding

	API (Restlet)	Representation (XML/JSON)	Model (Hibernate, Java EE)	Persistence (MySQL)
Item	Restlet resource bound to the URI. Supported default operations are GET, PUT and DELETE.	Representation parsing/generation based on the item attributes. Subresources inlined per request basis.	A native Java object (POJO) generated for each item with a Hibernate Data Access Object and binding to database elements.	Items are rows in respective database table with columns specified by item attributes. References map to foreign keys.
Container	Restlet resource bound to the URI. Supported default operations are GET and POST.	Representation parsing/generation delegated to Items.	Basic retrievals to database, using item mappings.	Containers are database tables.
Projection	Implemented on top of respective Containers.	Representation generation delegated to Container.	Extended retrievals to database, using item mappings.	Stored procedures for more advanced database queries. Tables implied by Container.

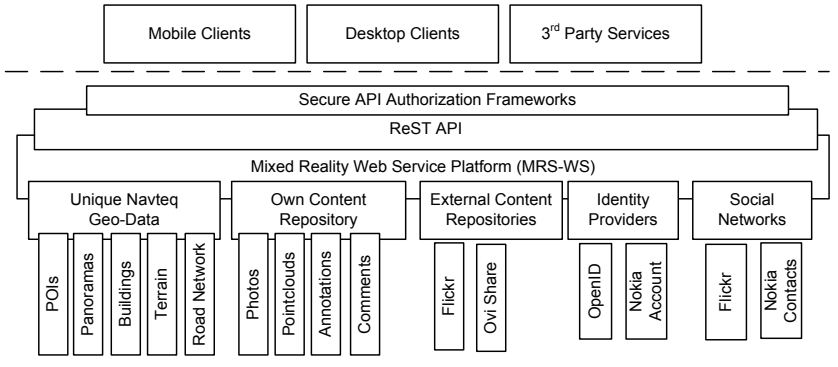

Fig. 3. Overview of the MRS Platform Architecture

To federate and provide aggregated contents, the platform defines a generic interface and a set of standard data models for general geo-referenced data. The platform aggregates the contents returned from the interfaces and transform them into the federated representations requested by the clients. The returned data is then transformed and merged by the aggregator as same as the interface approach. The advantage of this approach is to scale up the platform easily. Unlike the interface approach, which the business logic must be implemented by each every connector and integrated closely to the aggregator component inside the platform, the callback approach enables the implementation hosting outside the platform, typically residing on the third party controlled servers. This gives much flexibility and freedom to the third parties and eases the platform maintenance.

4 Examples of MRS-WS API

Following the discussion above, we give some concrete examples of MRS-WS APIs, starting with the Annotation API. MR annotations can be attached to a particular location or a building, and they can comprise a title and a textual description, as well as link to other content elements. Essentially an Annotation is a way of the users to annotate physical entities with digital information and share those with other users.

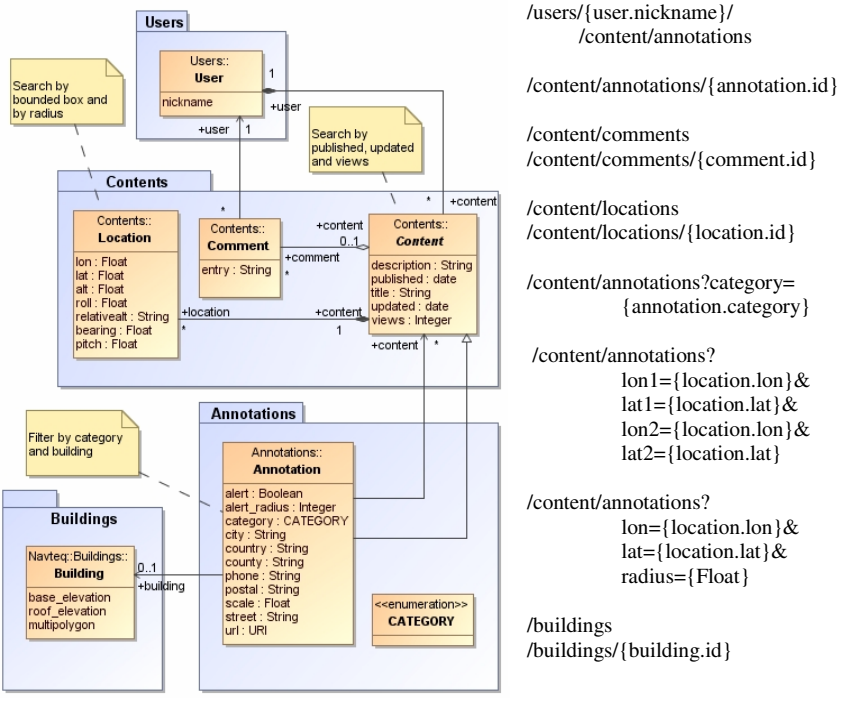

Fig. 4. Annotation Information Model and Implied Resource Hierarchy

Fig. 4 shows an Annotation information model, related resources and generated resource hierarchy. The default operations are as stated earlier: GET and POST for containers, GET, PUT, POST and DELETE for items and GET for projections. A sample representation subset for an annotation resource is given below, showing attributes inherited from Content and Annotation classes. Location subresources and Building reference are inlined to the representation, as shown in Fig. 5.

```
<annotation href="...">
    <id>4195042682</id>
    <updated>2009-12-18 04:01:13.0</updated>
    <published>2009-12-18 04:01:02.0</published>
    <title>I have an apartment for rent here!</title>
    <alert>true</alert>
```

```
    ...
    <locations>
      <location href="...">
        <lat>61.44921</lat>
        <lon>23.86039</lon>
         ...
        <pitch></pitch>
      </location>
    </locations>
    <building href="..">...</building>
</annotation>
```

Fig. 5. Example of content type application/vnd.research.nokia.annotation

4.1 Other MRS-WS APIs

The photo, video, audio and point cloud data objects can be accessed and modified via similar operations and representations. The API also offers access to other read-only geo-content originating from Navteq, such as

- building (/buildings) footprints and 3D models (Fig. 6. a & b);
- terrain (/terrain) tiles of earth's morphology (Fig. 6. c & d);
- street-view 360 panorama (/panoramas) photos (Fig. 6. e); and
- point-of-interest (/pois), with details about businesses and attractions.

To all those resources and containers, user-generated content and static geo-data, a uniform set of operations can be applied. Examples include:

- **geo-searching** for performing searches in a bounded box (e.g. /photos/?lat1=41.95&lon1=-87.7&lat2=41.96&lon2=-87.6) or in proximity (e.g. /photos /?lat=41.8&lon=-87.6&radius=1);
- **pagination** [x-mrs-api: page(), pagesize()] for controlling the number of objects fetched from a container per request by setting the page size and number;
- **verbosity** [x-mrs-api: verbosity()] for controlling the representation subset, ie.e selected attributes for the retrieved items; and
- **inlining** [x-mrs-deco: inline()] for selecting which subresources and referenced items are included per resources in the response to decide when to reduce the number of requests to the server and when to reduce the amount of data transferred.

Inlining is heavily used in MRS-WS for exposing advanced geo-data associations. For example, we have pre-calculated from the raw Navteq data the specific buildings that are visible in a given street view panorama, along with their associated POIs. A client requesting a specific panorama image can also request building data (e.g. links to 3D building models) to be inlined. This combined information can allow very advanced applications, such as touching and highlighting buildings, in an augmented reality view, for interacting with the real world (e.g. touch a building to find the businesses it hosts). Finally, when it comes to user-generated content, the following operations can be additionally applied:

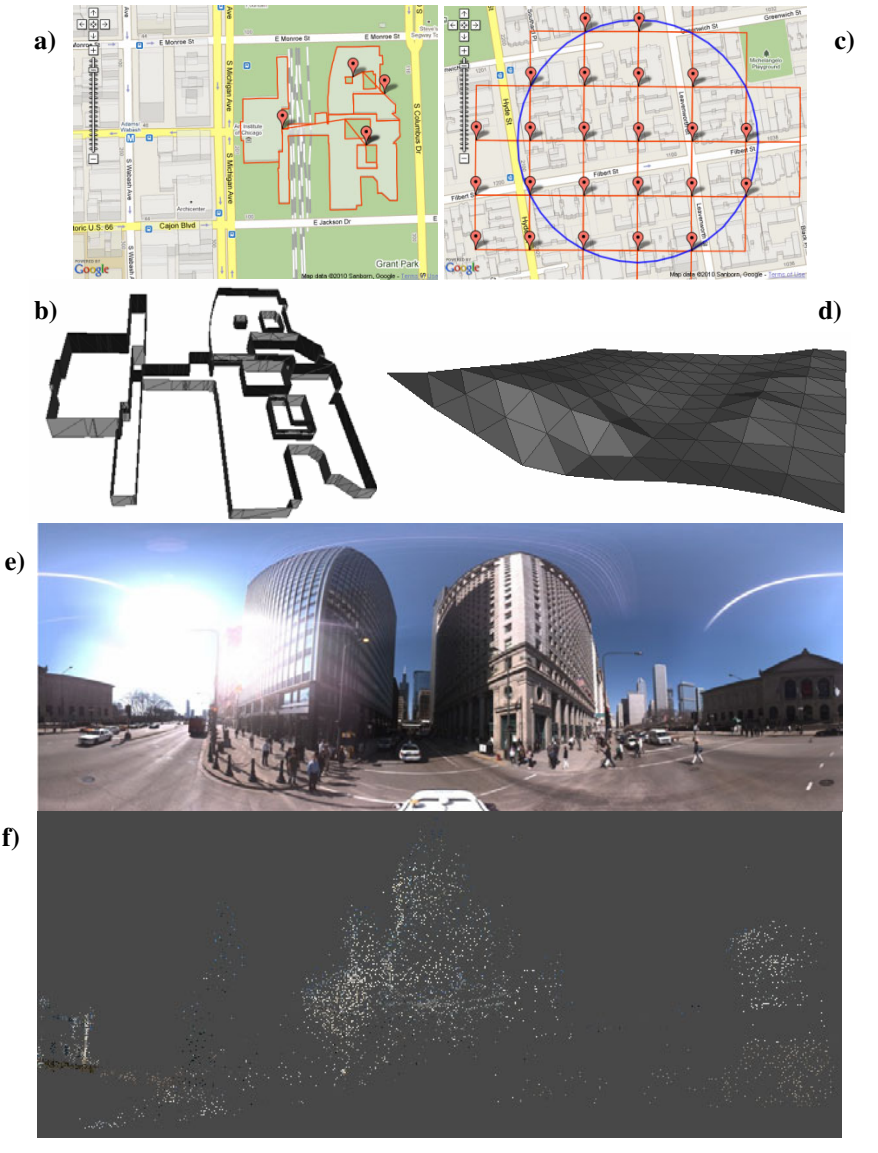

Fig. 6. Visualization of different data offered by our platform a) 2D footprint of a building, b) 3D model of a building, c) Searching terrain tiles in a given radius, d) 3D model of a terrain tile, e) Street-view 360 degree panorama, f) Point cloud of a church in Helsinki, created by multiple user photos

– **Temporal searching.** It is possible to limit the scope of the search within a specific time window. In the search query the time window (since- until) needs to be specified, in the form of UNIX timestamps. The matching is done against an item's "Updated time" or "Taken time", or combination of both (e.g. /photos/lat=41.8&lon=-87.6&radius=1&updated_since=1262325600)

- **Mixed Reality enabled.** Client can request results that explicitly include geo-spatial metadata, having in addition to geo-location at least associated bearing (yaw) metadata. This flag is set via the URI parameter mr_only=true.

5 Architectural Evaluation and Lessons Learned

Next, we revisit the non-functional requirements of the platform and review some of its other architectural quality attributes.

5.1 Non-functional Requirements Revisited

Usability defined as the ease of use and training of the service developers who build their clients against the platform. So far we have experience with several client teams building their applications against the platform and the feedback has been positive. We have not yet performed formal reviews of the usability, but we are currently preparing to adopt a 12-step evaluation process from Nokia Services as a medium for collecting feedback.

Modifiability interpreted as the ease with which a software system can accommodate changes to its software. Modifiability, extensibility and evolvability have proved to be the key architectural characteristics achieved through using our architecture. Integrating new requirements from the program clients has been very easy: the new requirements are integrated with the domain model, mapped to the resource model concepts, and to database schema and implementation level concepts. Based on our experiences, adding new concepts now takes around a few days. This is tightly related to the Resource Oriented Architecture style: mapping of the new concepts to the system is straightforward and supporting infrastructure for implementing functional requirements pre-exists.

Security and privacy as the ability to resist unauthorized attempts at usage, and ensuring privacy and security for user generated content. The platform supports a variant of Amazon S3 and OAuth for authorizing each REST request in isolation, instead of storing user credentials as cookies or with unprotected but obfuscated URIs. Our system has been audited for security by an independent body.

Scalability and performance in terms of the response time, availability, utilization and throughput behavior of the system. In the beginning of the program, we outlined goals for the scalability and performance in terms of number of requests, number of users, amount of data transferred, amount of content hosted and so forth. The main goal is to have an API and platform architecture that has built-in support for scalability and performance once the platform is productized and transferred to be hosted at business unit infrastructure. After initial performance tests with hosting a few million users and content elements, the system has so far been able to meet its requirements. While we have reason to believe that proper REST API, resource design and statelessness should be able to support scalability, for example through resource sharding, most performance characteristics like e.g. availability and utilization are not related to REST or Resource Oriented Architecture *per se* and are thus outside the scope of this paper.

Support for mobile devices is not one of the more common architectural quality attributes but is in the heart of the MRS-WS platform. By using the results of our previous work with REST and mobile devices, we support advanced content control for the clients through verbosity and inlining, and more effective bandwidth utilization through JSON and on-demand compression. Basically the clients can control the amount of data they want to receive and thus try to optimize the amount of requests and the amount of data being transmitted. It should be noted that optimizing for mobile clients includes a variety of topics, ranging from HTTP pipelining and multipart messages to intelligent device-side caching. Such techniques are outside the scope of this paper. We do claim, however, that the architecture we have chosen provides substantial support for implementing efficient mobile clients, and this is even more important in the domain of Mixed Reality services.

There are other system qualities, like conformance with standard Internet technologies and Nokia Services business unit technology stack, and still others that with a suitable interpretation are quite naturally supported by Resource Oriented Architecture like interoperability, one of the key promises of REST, but they are outside the scope of this paper. However, there is one additional quality that was not originally considered but that proved to be harder to achieve than initially expected: testability.

Testability interpreted as the ease with which platform can be tested. Testing complex platforms is *a priori* a hard task. It became obvious quite early in the development that as most resources share similar common functionality, changes to one part in the implementation can typically have effects throughout the API, making automated API level regression testing very desirable to ensure rapid refactoring. However, this is not currently properly supported by existing tools and frameworks. We have therefore built our own test framework for in-browser API level testing using JavaScript and XHR. The tests can be run individually, or using JSUnit as a way for running several test pages and reporting the results. The additional benefit is that whenever a new version of the platform is deployed, either locally, at the development server or at the alpha servers, the tests are deployed as well. Even though our test framework takes care of e.g. user authentication and basic content control through HTTP headers, in practice the browser environment still makes writing test cases more laborious than would be good. Therefore we unfortunately often fall back to using curl command line tool for doing initial tests.

5.2 Insights on Developing RESTful APIs and Resource Oriented Architecture

One of the main benefits of having ROA based architecture is that the RESTful API exposes concepts that are very close to the domain model, thus making understanding of the API in itself relatively easy. The uniform interface and uniformity of the searching and filtering operations across the platform makes the learning curve for different API subsets smooth. However, implementing ROA based services can be difficult for software architects lacking previous experience. Therefore having a lightweight process of building ROA services that is not tied to a particular implementation technology would be very useful for a project to be successful. We

also found out that while a REST API is in principle simple, programming against it requires a change in the programmer mindset from arbitrary APIs to a uniform interface with resource manipulations, which sometimes can be more challenging than expected.

Interoperability and its serendipitous form, ad-hoc interoperability, are hard to achieve without commonly agreed content types. One of the few ways of making interoperable systems is to use Atom, but without MR specific features it reduces the system into yet another content repository. Another topic for further research is how to evaluate a RESTful API against principled design characteristics like addressability, connectedness, statelessness and adherence to uniform interface. Issues like proper resource granularity, supporting idempotency, having one URI for one resource, and balancing between number of requests and amount of data are not commonly laid out. There is a clear lack of REST patterns and idioms, although the upcoming RESTful Web Services Cookbook [1] is a step towards the right direction.

5.3 Linking to Multiple Service Providers and Support for Multiple Sign-Ins

Generally users have multiple accounts on different services coming with different identity providers. Our service requires mashing up authorized external data from different services on behalf of the users. Instead of storing raw user credentials, our service saves the access tokens or opaque identifiers from external SPs to authenticate the requests on his behalf. Users can revoke the rights at the SP side easily to invalidate the access token, i.e. the authorized connection between our service and the SP. Additional benefit for the users is the ability to sign in with to their favorite identify provides without a need to create new passwords. The benefit of having multiple sign-in is to minimize the risk of identity service breakdown. The downside of the multiple sign-in is that our service has difficulties to maintain a least common denominator of user profiles.

5.4 Legal and Terms-of-Service Issues for a Service Platform

Our initial design goal was to create a generic platform on top of which our first lead service could be publically deployed. The subsequent services could later be deployed on top of the very same platform instance, allowing the users to be already registered and to potentially have pre-existing content, seamlessly accessible by the different clients, as simply different views or ways to explore the same data. However, it turned out that this approach, while very attractive engineering-wise, was legally considered very problematic. The Terms of Service (ToS) and the related Privacy Policy have to be very specific to the users on what they are registering for, and how the service is using their content. Asking the users to register for a single service now and later on automatically deploying more services, having their content automatically available, is against the policies for simple and clear ToS. One solution to the problem would be to explicitly ask the users to opt-in for the new services and get their permission to link their existing content, before making it available to the new service, which would also have its own ToS.

5.5 Related Work

While a relatively new topic, a number of books on the development of RESTful Web services have come about during the last year or are in the pipeline after the original [10] book came out. The upcoming RESTful Web Services cookbook [1] outlines recipes on the different aspects of building REST Web services. However, most of its recipes lean more towards identifying a design concern than giving a solution or stating the different forces related to using it. There are also books on how to implement RESTful Web services on particular frameworks like Jersey and Apache Tomcat and .NET. As one example, [3] gives a good if short overview on how to concretely implement RESTful Web services using Java and JAX-RS. The resource modeling approach described in this paper is an effort to outline the approach from REST and ROA point of view without any particular implementation platform in mind—in fact, it was originally developed and applied with Ruby on Rails.

6 Concluding Remarks

This paper described the development of a RESTful Web service platform for Mixed Reality services. We outlined our approach for developing RESTful Web services from requirements, summarized the platform architecture, evaluated some of its quality attributes and described insights gained during the development. We also presented some of the benefits identified from adhering to the Resource Oriented Architecture style and RESTful Web services, including having a close mapping from domain model to architecture in the case of content oriented systems, decoupling of clients and services, improved modifiability of the platform, and ability to support mobile clients.

We also encountered some challenges. While REST has become something of a buzzword in developing Internet Web services, there is a surprisingly small amount of real-life experience reports available concerning the development of services conforming to the Resource Oriented Architecture. In particular, there is a lack of modeling notations and methods for systematically developing RESTful services. The team also had to develop its own testing tools as no suitable ones existed.

As future work, the team will continue developing the platform for future clients to be released during spring 2010. From a Web services point of view, the team plans to continue research on the RESTful Web services engineering while from a Mixed Reality point of view, the focus will be on supporting more advanced contextuality, providing related and relevant information through associative browsing, and orchestrating content and metadata coming from different sources.

Acknowledgments. The authors wish to thank Arto Nikupaavola, Markku Laitkorpi, the former NRC Service Software Platform and the NRC Mixed Reality Solutions program for their valuable contribution.

References

1. Allamaraju, S., Amudsen, M.: RESTful Web Services Cookbook. O'Reilly, Sebastopol (2010)
2. Azuma, R.: A Survey of Augmented Reality. Presence: Teleoperators and Virtual Environments 6(4), 355–385 (1997)
3. Burke, B.: RESTful Java with JAX-RS. O'Reilly, Sebastopol (2009)
4. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Doctoral Thesis, University of California, Irvine (2000)
5. Föckler, P., Zeidler, T., Brombach, B., Bruns, E., Bimber, O.: PhoneGuide: museum guidance supported by on-device object recognition on mobile phones. In: 4th international Conference on Mobile and Ubiquitous Multimedia MUM '05, vol. 154, pp. 3–10. ACM, New York (2005)
6. Höllerer, T., Wither, J., DiVerdi, S.: Anywhere Augmentation: Towards Mobile Augmented Reality in Unprepared Environments. In: Location Based Services and TeleCartography. Lecture Notes in Geoinformation and Cartography. Springer, Heidelberg (2007)
7. Laitkorpi, M., Selonen, P., Systä, T.: Towards a Model Driven Process for Designing RESTful Web Services. In: International Conference on Web Services ICWS '09. IEEE Computer Society, Los Alamitos (2009)
8. Milgram, P., Kishino, F.: A Taxonomy of Mixed Reality Visual Displays. IEICE Transactions on Information Systems E77-D (12), 1321–1329 (1994)
9. Reitmayr, G., Schmalstieg, D.: Location based applications for mobile augmented reality. In: Biddle, R., Thomas, B. (eds.) Fourth Australasian User interface Conference on User interfaces 2003 - Volume 18, Adelaide, Australia, vol. 18. ACM International Conference Proceeding Series, vol. 16, pp. 65–73. Australian Computer Society, Darlinghurst (2003)
10. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Sebastopol (2007)