# RESTful, Resource-Oriented Architectures: A Model-Driven Approach

Sandy Pérez[1], Frederico Durao[2], Santiago Meliá[3], Peter Dolog[2], and Oscar Díaz[1]

[1] ONEKIN Group, University of the Basque Country, San Sebastián, Spain
{sandy.perez,oscar.diaz}@ehu.es
[2] IWIS Group, Aalborg University, Aalborg, Denmark
{fred,dolog}@cs.aau.dk
[3] IWAD Group, University of Alicante, Alicante, Spain
santi@dlsi.ua.es

**Abstract.** RESTful Web services have opened the door to clients to use Web sites in ways the original designers never imagined giving rise to the mashup phenomenon. The main advantage of the model based approach in Web engineering is that the models specify sort of contract the Web application adheres to and promises to deliver. Similarly, in RESTful scenario, mashup components responsible for delivering composite functionalities out of RESTful components could benefit from such contracts in search, automatic mashup, and other scenarios. Such scenarios ground the need for taking RESTful Web services in existing Web methods. This paper proposes the *Application Facade Component Model* in existing Web methods to support RESTful, resource-oriented architectures generation. Amazon Simple Storage Service is used as the running example and proof of concept to show advantages of such approach.

**Key words:** rest, restful, resource-oriented, model-driven, Software-as-a-Service.

## 1 Introduction

Web 2.0 has brought the Web-as-a-platform movement whereby Web applications open their data silos for others to capitalize upon and can act as services. This implies the need for Application Programming Interfaces (APIs) in order to make Web applications accessible for mashing up over the Web. That is, Web applications can be accessed as services over Internet and executed on a remote system hosting the requested services. Nowadays, according to Programmableweb.com on 22/06/2010, the 72% of APIs available on the Web follow the REST-style architecture.

To this end, mashup research concentrated mostly on developing engines, methods, and models for composing from available RESTful components (see for example [9,15,11]). However, any composition engine or service discovery components rely on metadata about components to be considered in such compositions. To our knowledge, very little effort has been invested to easy creation of such metadata.

Traditionally, Web service architectures follow the RPC-style. Commonly, this tends to suggest that offerings are classified in terms of *verbs* (e.g. borrowing, buying and

D.K.W. Chiu et al. (Eds.): WISE 2010 Workshops, LNCS 6724, pp. 282–294, 2011.

the like) while nouns play the role of parameters. By contrast, REST, an alternative architectural style, works the other way around: offerings tend to be classified in terms of *nouns* (e.g. book) while verbs qualified the nouns. Nouns will then stand for *resource*. A *resource* can be essentially any coherent and meaningful concept that may be addressed (e.g. a document or an image, a collection of other *resources*, the result of an algorithm, etc.).

On the other hand, data-intensive Web applications are particularly well-fitted for REST-style architectures. Indeed, proposed methods for data-intensive Web applications are built around the so-called "*Domain Model*". A *Domain Model* captures the main entities and relationships found in the application domain. This model is then complemented with other models that capture additional perspectives of the applications, namely, the *Navigational Model* (which specifies the data to be presented, as a view of the *Domain Model*, and the order in which this data is to be presented) and the *Presentation Model* (i.e. a static representation of the widgets as structural components of a view). The bottom line is that REST-style architectures and data-intensive applications put *Domain/Resource Models* at the very centre of their design.

This paper surfaces such parallelism by specifying how RESTful interfaces can be derived from data-intensive application models. From this perspective, this work aligns with current efforts to model-driven Web application generation. Our approach complements previous process (see [4] for an overview). For familiarity reasons, examples follow the OOH4RIA model notation [7].

Therefore the main contributions of this paper are:

- providing a model for the application facade component, i.e. the end point for requesting the application services;
- a set of QVT model transformations which map *Domain/Navigational Models* into the RESTful interface counterparts.

The main outcomes of the transformations include generating the Universal Resource Identifiers (URIs) [1], path and query parameters, HTTP headers, and so on. Amazon Simple Storage Service (Amazon S3) is used as the running example.

The rest of the paper is structured as follows. Section 2 provides basic background on REST and RESTful, resource-oriented architectures. Section 3 introduces the running example—the Amazon S3. Section 4 presents the OOH4RIA's *Domain Model* and *Navigational Model* with the help of the running example. The main contribution of the paper rests on section 5 that introduces the *Application Facade Component Model* (AF-CM). Section 6 outlines the relevant related work. Finally, some conclusions end the paper.

## 2 Background

REpresentational State Transfer (REST) is an architectural style for distributed hypermedia systems [5]. REST-style architectures consist of clients and servers. Clients make requests to servers and servers respond to their clients by acting upon each request and returning appropriate responses. Requests and responses are built around the transfer of *representations* of *resources*. A *resource* can be essentially any coherent and meaningful concept that may be addressed (i.e. the intended conceptual target of a hypertext

reference). However, servers cannot send a "concept" to their clients: they send a series of bytes in a specific file format (e.g. an XML document or a comma-separated text). This is a *representation* of a *resource*. A *representation* is just some data about the current state of a *resource*. *RESTful Web services* are simple Web services implemented using HTTP and the principles of REST. They expose standard HTTP objects (i.e. *resources*) that respond to one or more of the six[1] standard HTTP methods: GET, HEAD, POST, PUT, DELETE, and OPTIONS.

### 2.1 RESTful, Resource-Oriented Architectures (ROAs)

ROA as documented by Leonard Richardson and Sam Ruby [10] is a specific set of guidelines of an implementation of the REST-style architecture. ROAs are based on four concepts:

1. Resources (e.g. the article about REST in the Wikipedia).
2. Their names (URIs). The URI is the name and address of a resource. For example, http://www.wikipedia.org/wiki/Representational_State_Transfer.
3. Their representations. A resource is a source of representations.
4. The links between them. Normally a hypermedia representation of a resource contains links to others resources.

and four properties:

1. Addressability. Addressable applications expose a URI for every piece of information they might conceivably serve. It makes easy for clients to use Web sites in ways the original designers never imagined.
2. Statelessness. Statelessness means that every HTTP request happens in complete isolation. The server never relies on information from previous requests.
3. Connectedness. A Web service is connected to the extent that you can put the service in different states just by following links and filling out forms.
4. A uniform interface. In ROAs, HTTP is the uniform interface. GET method to retrieve a representation of a resource, PUT method to a new URI or POST method to an existing URI to create a new resource, PUT method to an existing URI to modify a resource and DELETE method to remove an existing resource. Probably HTTP methods are not a perfect interface but what is important is the uniformity [10]. The point is not that GET is the best name for a read operation, but that GET means "read" across the Web. Given a URI of a resource, everybody knows that to retrieve the resource s/he has to send a GET request to that URI.

## 3 A Running Example: Amazon S3

Amazon S3 provides a simple Web service interfaces that can be used to store and retrieve any amount of data, at any time, from anywhere on the Web. We can keep our

[1] Actually there are eight standard HTTP methods. However, CONNECT and TRACE are rarely used when implementing RESTful Web services.

**Table 1.** Some operations from the Amazon S3 REST API

| | **Service** | **Bucket** | **Object** |
|---|---|---|---|
| **GET** | Lists all *buckets* owned by the authenticated sender of the request.<br><br>GET / HTTP/1.1<br>Host: s3.amazonaws.com<br>Authorization: signature | Lists some or all of the *objects* in a *bucket*.<br><br>GET / HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature | Retrieves *object* from Amazon S3<br><br>GET /object HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature |
| **HEAD** | | | Retrieves metadata from an *object* without returning the *object* itself.<br><br>HEAD /object HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature |
| **PUT** | | Creates a new *bucket*.<br><br>PUT / HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Content-Length: 0<br>Authorization: signature | Adds an *object* to a *bucket*.<br><br>PUT /object HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature<br>Content-Type: text/plain<br>Content-Length: 11434<br>Expect: 100-continue<br>[11434 bytes of object data]<br><br>- or -<br><br>Creates a copy of an *object* that is already stored in Amazon S3.<br><br>PUT /destObj HTTP/1.1<br>Host: destBckt.s3.amazonaws.com<br>x-amz-copy-source: /bucket/object<br>Authorization: signature |
| **POST** | | | Adds an *object* to a specified *bucket* using HTML forms.<br><br>POST /object HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature<br>Content-Type:<br>multipart/form-data;<br>boundary=9431149156168<br>Content-Length: length<br>–9431149156168 |
| **DELETE** | | Deletes the *bucket* named in the URI.<br><br>DELETE / HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature | Removes the *object*.<br><br>DELETE /object HTTP/1.1<br>Host: bucket.s3.amazonaws.com<br>Authorization: signature |

data private, or make it accessible by anyone with a browser. However, for the sake of simplicity, we will get focused only on the Web service interfaces leaving the security aspects out of this paper.

The key Amazon S3 concepts are: *objects*, *buckets*, and *key*. *Objects* consist of object data and metadata and they are the fundamental entities stored in Amazon S3. A *bucket* is a named container for *objects*. A *bucket* is analogous to the file system on our hard drive, and an *object* to one of the files on that file system. Finally, a *key* is the unique identifier for an *object* within a *bucket*. Together, a bucket name and a *key* uniquely identify an *object* in Amazon S3. For example, in http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl, "doc" is the bucket name and "2006-03-01/AmazonS3.wsdl" is the *key*.

Table 1 describes some operations of the real REST API offered by Amazon S3. For the sake of understandability we chose only those operations we think they will help us to explain our approach keeping things as simple as possible. In table 1, columns represent resources whereas rows represent the HTTP methods. Each cell represents the effect of calling the corresponding HTTP (i.e. row) over the corresponding resource (i.e. column).

## 4 Modelling Amazon S3 with OOH4RIA

OOH4RIA [8] proposes a RIA-specific model-driven development (MDD) process based on a set of models and transformations to obtain the implementation of Rich Internet Applications (RIAs). This approach specifies an almost complete RIA through the extension of the OOH server-side models (i.e. *Domain* and *Navigational*) with two new models (i.e. *Presentation* and *Orchestration*). Recently, this approach has been extended by introducing the *Feature* and *Component Models* that represent the technological and architectural RIA variability. Following the Amazon S3 sample, we start defining the application functional models that permit us to represent the problem-space elements situated at the server side. The OOH4RIA defines two DSL (Domain-Specific Language) server models: the *Domain Model* and the *Navigational Model*. The *Domain Model* represents the most important domain entities, free from any technical or implementation details and, on the other hand, the *Navigational Model* constraints how the client side can navigate through the most relevant semantic paths querying and filtering the domain elements (entities, attributes, operations and queries) that will be offered to the client side. In fact, the *Navigational Model* permits us to establish a specific interface offered by the server side to the client side.

Figure 1 shows how the *Domain Model* represents different entities and their relationships of the Amazon S3 sample. An Amazon S3 *User* manages a collection of 100 *Buckets* where each of them contains a set of *Objects*. As is stated before, an *Object* represents a file with its data and metadata. *Users* can *create* or *remove Buckets* and *Objects* as well as *copy Objects*. At this point, the designer must define the Amazon S3 *Navigational Model* in order to define the server side interface. The navigation starts when a *User NavigationalClass* gets all its *Buckets* (i.e. *listAllMyBuckets* "*traversal link*" with the <<*showAll*>> stereotype). Once these *Buckets* are gathered a *User* can *create* or *delete* a *Bucket* or obtain its *Objects* using a paging mechanism (i.e. *listBucket*

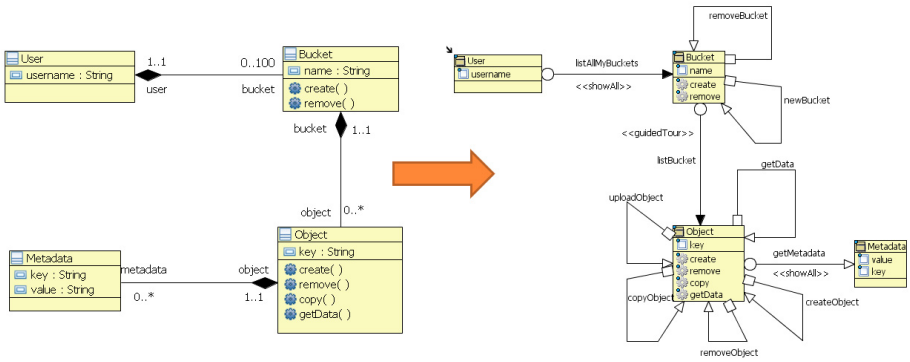


**Fig. 1.** The *Domain* and *Navigational Models* of Amazon S3

*traversal link* with the <<*guidedTour*>> stereotype). Once the navigation is situated in the *Object NavigationalClass* a *User* can execute a set of "*service links*" to manage these *Objects* such as *createObject*, *removeObject*, *uploadObject* and *copyObject*. Moreover, he can obtain the *Object*'s binary data using the *getData service link* or *Object*'s metadata that describes it (i.e. *getMetadata traversal link*).

These problem-space server models represent the Amazon S3 entities and how we can navigate through them. At this point, we must define solution space artefacts that specify explicitly which is the architecture and technologies that we must use in a specific case.

## 5 Modelling REST's Concerns

In this work, we extended OOH4RIA defining new artefacts and activities that permit us to obtain a RESTful, resource-oriented architecture at server side. REST and RPC-style are just the interface that the server offers to clients to allow them to interact with the business logic. Independently if our server exposes a REST or RPC-style interface, the business logic of our application (i.e. *Domain Model* and *Navigational Model*) should remain being the same. So, where can we introduce REST specificities?

### 5.1 The RIA Component Model

In [8], we have introduced a new model—the *RIA Component Model* (RIA-CM), which features an explicit representation of RIA architecture. The RIA-CM is a component-based architectural style that represents a structural view of RIA applications. This model defines a component topology, with each component representing a role or task performed by one or more common components identified in the RIA family.

Figure 2 shows the RIA-CM for Amazon S3 example. In this model, the <<*ApplicationFaçade*>> is the component responsible for offering the services that the client side can invoke to perform a certain task. It is in this model where REST specificities

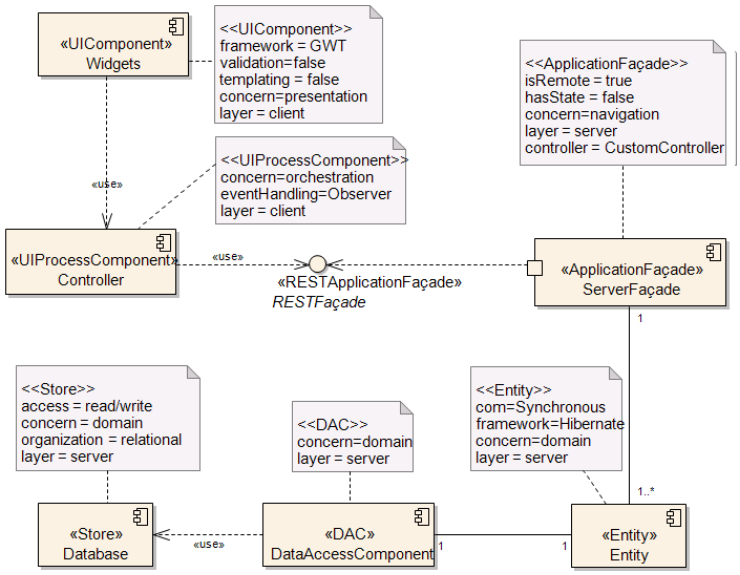


**Fig. 2.** The *RIA Component Model* (RIA-CM) for the Amazon S3 sample

are going to be taken into account, more specifically by the <<*ApplicationFaçade*>> component.

As we can see in figure 2, the <<*ApplicationFaçade*>> component offers a *RESTApplicationFaçade* (i.e. a RESTful interface). Similar to RPC-style, this interface contains operations coming from the *Navigational Model*. However, with the RPC-style these operations are offered directly as they are in the *Navigational Model* whereas the REST-style requires these operations to be offered through a uniform interface (i.e. HTTP methods). That is, operations from the *Navigational Model* need to be mapped to HTTP methods. But, in figure 2 operations are not available. So, how can we establish the mapping?

### 5.2 The Application Facade Component Model

To visualize operations, we propose a new model—the *Application Facade Component Model* (AF-CM). In our environment this model is accessible through a double click on the <<*ApplicationFaçade*>> component. It is inspired by the white-box view of the internal structure of a component that contains other components in UML. It allows a *Software Architect* to take a look into the internal structure of the <<*ApplicationFaçade*>> component. View includes the offered interfaces with their operations and parameters (see figure 3).

**Resources.** Any *NavigationalClass* with at least one outgoing *traversal link* or *service link* at the *Navigational Model*, it is a candidate to became a resource in the AF-CM. In figure 1 only *User*, *Bucket* and *Object* have *traversal links* (the arrow with a circle

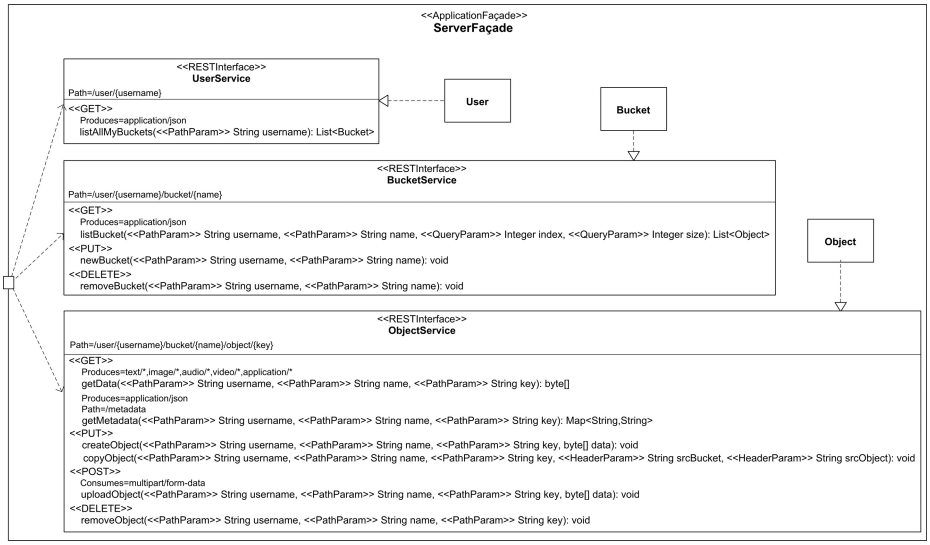


**Fig. 3.** An *Application Facade Component Model* for the Amazon S3 sample

in the source) or *service links* (the arrow with a square in the source) and *Metadata* do not have any. *User*, *Bucket* and *Object* are candidates to become resources in the AF-CM but not *Metadata* (see figure 3). Notice that every resource in the AF-CM offers an interface containing one operation per *traversal/service link*.

**A uniform interface.** The mapping between operations in the *Navigational Model* and HTTP methods is defining by annotating the operations in the interfaces owned by the AF-CM. Operations inside these interfaces are actually proxies of the operations defined in the *Navigational Model*. Annotations are based on the "The Java API for RESTful Web Services" (JAX-RS) specification [12]. Operations are grouped, depending on their mappings, according to an HTTP method, that is, inside an interface we will have as many operations groups as HTTP methods.

**URIs.** As you can note in figure 3, every interface has a path assigned to it. That path will be appended to the Web application URL and the result will become the URI of the resource that realizes such interface. For example, the resource *User* could have http://www.mydomain.com/myapp/user/{username} as URI, where {username} is a path parameter corresponding to the *User*'s username (i.e. the attribute that plays the role of primary key). In other words, every *User* will have a different path. In the case of *Buckets* and *Objects*, things change a bit. *Buckets* can be uniquely identified as owned by a *User* since there is a composite aggregation (i.e. a whole/part relationship) between them where *User* is the composite object (see *Domain Model* in figure 1). Therefore, *Bucket*'s path must include the *User* owning the *Bucket*. The same occurs with *Objects* and *Buckets*. Moreover, operations can have a path, that is, operations can be offered as resources by themselves. That is the case of *getMetadata* operation in the interface realized by *Object*, which will have the "http://www.mydomain.com/myapp/user/{username}/bucket/{name}/object/{key}/metadata" as URI.

**Representations.** Servers send to client representations of resources. For example, the list of *Buckets* returned by *listAllMyBuckets* operation in the interface realized by *User* could be represented as an XML document or as a list following the JSON format. To support the different representations a server can send to clients we use the "Produces" annotation. Suppose *listAllMyBuckets* operation returns a list following the JSON format, we simply have to annotate that as "Produces=application/json". However, representations can flow the other way, too (e.g. clients can send a representation of a new *Object* to the server and have the server create the *Object*). In this case, the corresponding annotation is "Consumes". For example, in figure 3, *uploadObject* operation in the interface realize by *Object* is annotated as "Consumes=multipart/form-data" that means the request's document corresponds to a HTML form submission.

**Parameters**. Parameters' values can be sent in different ways: placed in the path as part of the URI (i.e. <<*PathParam*>>), placed in the query, after the '?' character in the URI (i.e. <<*QueryParam*>>), as a HTTP header (i.e. <<*HeaderParam*>>), in the cookie (i.e. <<*CookieParam*>>) or as a form field (i.e. <<*FormParam*>>). One example: the *copyObject* operation in the interface realized by *Object* that creates a new *Object* as a copy of an existing one. *copyObject* takes the *username*, the *name* of the *Bucket* inside which the new *Object* will be created and the *key* of the new *Object* (recall *User*, *Bucket* and *Object*'s *key* identify an *Object* uniquely) from the path as <<*PathParam*>> whereas the *name* of the *Bucket* where the exiting *Object* resides and the *key* of the existing *Object* are read as values of HTTP headers (i.e. <<*HeaderParam*>>), see figure 3.

### 5.3 Model-to-Model Transformation

However, the *Application Facade Component Model* (AF-CM) is not created from scratch. A first skeleton is generated from the *Domain Model* and the *Navigational Model* which is later enriched and/or modified by the *Software Architect*. To this end, our approach includes a set of QVT transformations. Figure 4 shows an example—the *CreatingServiceLink2OperationPUT* QVT rule owned by the *Nav2RESTInterface* transformation.

The *CreatingServiceLink2OperationPUT* QVT rule converts a *service link* of the *Navigational Model* into an operation of the AF-CM. On the left side, the rule checks that exists a *service link* that departs from a source *NavigationalClass* has a not auto-generated *OID* (Object IDentifier) attribute (i.e. its *isAuto* value is false) since a *NavigationalClass* with an auto-generated *OID* is created using the POST HTTP method instead of PUT. The rule also verifies that the operation enacted by the *service link* is a "create" operation in the *Domain Model*. At this point, it is important to note that in the *Domain Model* operations can be marked as being a CRUD method (i.e. *create*, *read*, *update* and *delete*). In other words, it checks if the value of *opType* attribute is *create*. If the left side is accomplished, the rule creates a new operation in the corresponding interface at the AF-CM stereotyped as *PUT* whose name corresponds to the *service link*'s name (i.e. *nsl*). Moreover, the rule creates a <<*PathParam*>> parameter per parameter in the interface's path. Finally, the rule invokes the *SLParameterToQueryParam* that generates a new <<*QueryParam*>> parameter per parameter in the operation enacted by the *service link*. Similar transformations are defined for the rest of CRUD methods. Table 2 shows how CRUD methods are mapped into HTTP methods.

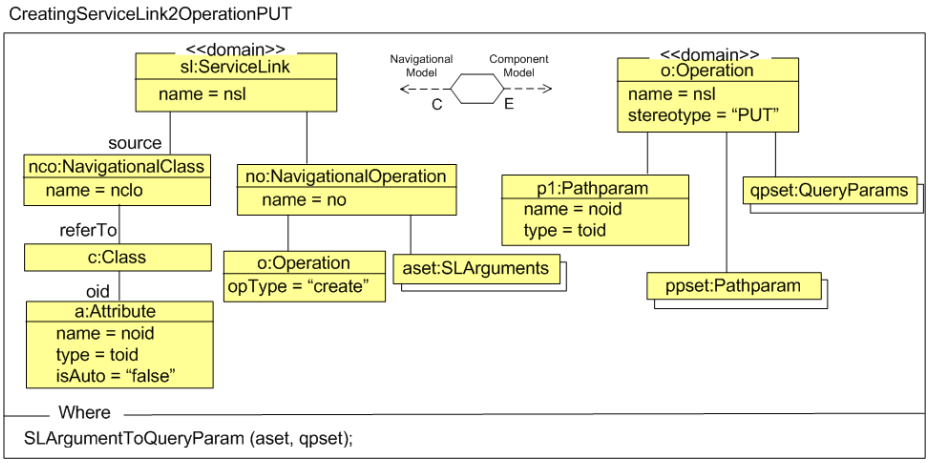


**Fig. 4.** The QVT Relation *CreatingServiceLink2OperationPUT* of the *Nav2RESTInterface* Transformation

**Table 2.** Mapping CRUD methods into HTTP methods

| **CRUD Method** | **HTTP Method** |
|---|---|
| create (OID typed by the user) | PUT |
| create (auto-generated OID) | POST |
| read | GET |
| update | PUT |
| delete | DELETE |

The generated skeleton of AF-CM can then be edited by the *Software Architect*. For example, by adding operations not marked as being a CRUP method in the *Domain Model* such as the *copy* operation of the *Object* class, which is enacted by the *copyObject service link* in the *Navigational Model*.

## 6 Related Work

The Web engineering community is well-aware about the importance of Web services and much work has been done in this area, [3,2,13] are just an example. However, surprisingly there is few work related to RESTful Web services. To the best of our knowledge, only OOWS and WebML deal with RESTful Web services in some way.

**OOWS.** In [14], authors present a model-driven approach to integrate existing REST APIs into model-driven Web engineering methods. They propose an approach where resources are modelled as entities of the *Domain Model*. To this end, authors provide a REST metamodel whose main goal is to describe RESTful services. Their approach is aimed to create Web applications which are able to interact with existing REST APIs (e.g. Amazon S3) but not to create the Amazon S3 application, which is the goal of our approach.

**WebML.** WebRatio Site Development Studio is a commercial Web design tool suite that implements the WebML modelling language and its accompanying design method. According to WebRatio's wiki[2], it is possible to publish a Web service that uses the REST style invocation from WebML models. The URL of the generated Web services is composed as follows:

http://<host>:<port>/<webApplication>/<ServiceViewName>/<PortName>/
<*SolicitName*>.do? <SolicitParameterName1>=<value1>

For example:

http://localhost:8080/AcmeWS/WS_Publish/Operations/*getProductsByName*.do?
keyword=Allair

Notice that *method information* (i.e. *getProductsByName*) is included in the URL. Some people classify these REST services as low REST services. Low REST services tend to deviate from orthodoxy in a particular direction (toward the RPC style), in other words, they follow a REST-RPC hybrid architecture. On the other hand, our approach is aimed to support high REST services generation. High REST services are just those that adhere closely to the Fielding dissertation [5].

In mashups community, most of the work reports on composition approaches but does not deal with the fact how to ease the creation of RESTful component suitable for composition or mashup. [9] looks at how the RESTful components can be provided and used within standardized business BPEL specifications. It argues that with WSDL 2.0 one can achieve some correlations to allow for composing with BPEL but also with some limitations. [15] argues for development of tools which change the way how mashups are created out of the content and services available on the Web. [11] argues for a lightweight workflow language which would allow dynamic and flexible composition of services available on the Web. Our approach provides a missing link for the aforementioned composition approaches: it provides the metadata generated out of the Web application design models which can be used by the composition approaches and languages.

Closer approaches to the present work include [6]. It presents a model-driven process for designing RESTful Web services. The analysis phase produces a functional specification that captures the relevant interactions between the service and its clients. The functional specification is then transformed into an information model, which captures the behaviour of the intended API. The next step in the process transforms the information model into externally visible, interconnected resource entities as a resource model. Finally, the resource model is interpreted according to the target architecture and produces output that is concrete enough to be used as input for implementation tools and service frameworks. However, unlike [6], our approach is not RESTful specific. The generated server side code can follow a RESTful, resource-oriented or a RPC architecture. On the other hand, [6] uses nomenclature which is not close to Web service developers, in our case we opted for using nomenclature coming from JAX-RS specification [12] which could be more familiar to RESTful Web service developers.

[2] http://wiki.webratio.com/index.php/Getting_started_with_Web_Services

However, we must not forget that much of the success of REST is due to its widespread use in business. Nowadays, top integrated development environments offer tools to make the development of RESTful Web service easier. For example, the NetBeans[3] IDE can generate RESTful Web services from existing entities classes. However, current IDEs solutions work at the code level.

## 7 Conclusions

This paper proposes the introduction of the *Application Facade Component Model* in existing Web methods to support RESTful, resource-oriented architectures. The use of an MDD approach accounts for facing in a stepwise manner the different issues risen during the development of a Web application, mainly, entities and relationships found in the application domain, the data to be presented and the order in which this data is to be presented, the architectural configuration of the application at hand, and so on. These decisions are decoupled from the chosen technological platform. As a proof-of-concept, an MDD process is defined using OOH4RIA metamodels as PIMs. QVT and Xpand are used as the model-to-model and model-to-code transformation languages.

Future work includes the introduction of security concerns in the approach. Also, we plan to address how to integrate existing RESTful APIs (e.g. Twitter, Facebook, YouTube, etc.) into current Web methods. And the last but not least, some experiments are planned in the near future.

**Acknowledgments.** This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2008-06507-C02-01/TIN (MODELINE) and TIN2007-67078 (ESPIA), the Avanza I+D initiative of the Ministry of Industry, Tourism and Commerce under contract TSI-020100-2008-415, the project "KiWi - Knowledge in a Wiki" and is partly financed by the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement No. 211932. Pérez enjoys a doctoral grant from the Basque Government under the "Researchers Training Program".

## References

1. Berners-Lee, T.: Universal Resource Identifiers – Axioms of Web Architecture (December 1996), Published at `http://www.w3.org/DesignIssues/Axioms`
2. Brambilla, M., Ceri, S., Comai, S., Fraternali, P., Manolescu, I.: Model-driven Specification of Web Services Composition and Integration with Data-intensive Web Applications. IEEE Data Engineering Bulletin 25(4), 53–59 (2002)
3. Bruni, R., Hölzl, M., Koch, N., Lafuente, A.L., Mayer, P., Montanari, U., Schroeder, A., Wirsing, M.: A Service-Oriented UML Profile with Formal Support. In: 7th International Conference on Service Oriented Computing (ICSOC/ServiceWave 2009) (2009)
4. Escalona, M.J., Koch, N.: Requirements Engineering for Web Applications - A Comparative Study. Journal of Web Engineering 2(3), 193–212 (2004)
5. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)

[3] http://netbeans.org/kb/61/websvc/rest.html

6. Laitkorpi, M., Selonen, P., Systä, T.: Towards a Model-Driven Process for Designing ReSTful Web Services. In: IEEE 7th International Conference on Web Services, ICWS 2009 (2009)
7. Meliá, S., Gómez, J., Pérez, S., Dáz, O.: A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In: 8th International Conference on Web Engineering (2008)
8. Meliá, S., Gómez, J., Pérez, S., Díaz, O.: Architectural and Technological Variability in Rich Internet Applications. IEEE Internet Computing 14(3), 24–32 (2010)
9. Pautasso, C.: Restful web service composition with bpel for rest. Data Knowl. Eng. 68(9), 851–866 (2009)
10. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Inc., Sebastopol (2007)
11. Rosenberg, F., Curbera, F., Duftler, M.J., Khalaf, R.: Composing restful services and collaborative workflows: A lightweight approach. IEEE Internet Computing 12(5), 24–31 (2008)
12. Sun Microsystems, Inc. JAX-RS: Java$^{TM}$ API for RESTful Web Services (2008)
13. Torres, V., Pelechano, V., Pastor, O.: Building Semantic Web Services Based on a Model Driven Web Engineering Method. In: International Workshop on Conceptual Modeling of Service-Oriented Software Systems, CoSS 2006 (2006)
14. Valverde, F., Pastor, O.: Dealing with REST Services in Model-driven Web Engineering Methods. In: V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB 2009 (2009)
15. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. IEEE Internet Computing 12(5), 44–52 (2008)