How do we apply the concept of resource orientation by designing representations to support interactions?

BY ERIK WILDE AND ROBERT J. GLUSHKO

Document Design Matters

THE CLASSICAL APPROACH to the data aspect of system design distinguishes *conceptual*, *logical*, and *physical* models. Models of each type or level are governed by metamodels that specify the kinds of concepts and constraints that can be used by each model; in most cases metamodels are accompanied by languages for

describing models. For example, in database design, conceptual models usually conform to the *Entity-Relationship (ER)* metamodel (or some extension of it), the logical model maps ER models to relational tables and introduces normalization, and the physical model handles implementation issues such as possible denormalizations in the context of a particular database schema language. In this modeling methodology, there is a single hierarchy of models that rests on the assumption that one data model spans all modeling levels and applies to all the applications in some domain.^a

The “one true model” approach assumes homogeneity, but that does not work very well for the Web. The Web as a constantly growing ecosystem of heterogeneous data and services has challenged a number of practices and theories about the design of IT landscapes. Instead of being governed by “one true model” used by everyone, the underlying assumption of top-down design, Web data and services evolve in an uncoordinated fashion. As a result, a fundamental challenge with Web data and services is matching and

happens for performance optimization reasons. In this case, the lower-level models are derived from the conceptual model, but they are also based on different assumptions about access patterns and required performance.

^a This simplified view may change if different implementations of the same conceptual model use different lower-level models, per-

mapping local and often partial models that not only are different models of the same application domain, but also differ, implicitly or explicitly, in their associated metamodels.

This challenge is central in the design of interfaces for machine-to-machine communications over the Web because in the general case, no entity can authoritatively and unilaterally define the interfaces and data models or choose the metamodel that describes them.^b On the Web, when peers use different conceptual models internally, they still can collaborate by using an intermediate model as a representation for their communications. This requires that each peer has a way of mapping its own conceptual model to the intermediary model's conceptu-

al model, which is then serialized according to the representation defined for that model, and then subsequently parsed, instantiated, and mapped to the other peer's conceptual model.

This description may sound as if the intermediary model now is the "one true model" mentioned earlier. The important difference to the assumption that there is an implicit model in all collaborating applications is that the Web approach focuses on exchange-oriented *resource representations* and employs those as the way how data models are represented for communications. However, this article is not about the concept of resource-orientation in the abstract (see Prescod⁴ for a good discussion of the fundamental differences of approaches); it is about the question of how to properly apply this concept by designing representations for the specific purpose of supporting interactions.

Somewhat paradoxically, mapping and serializing models is made more difficult by the flexibility and ubiquity

of the Extensible Markup Language (XML). Many people assume that merely by providing XML-based representations, universal interoperability and loose coupling can be achieved almost magically⁵—even though different metamodels that simply have some XML representation may make it almost impossible to access the conceptual information in such an XML serialization by using XML tools alone. In many cases, this exaggerated expectation of what XML can do is caused by mixing modeling layers, for example by assuming that *any* data serialized as XML can be conveniently processed using XML tools.

Without further qualification, the argument that "we use XML, which is a widely accepted data format, and thus our interface is easy to use" makes about as much sense as "we use bits, which are a widely accepted data format, and thus our interface is easy to use." On a simplistic technical level, these statements are true, but it is essential to understand the assumptions about application-level models and required tools that are implicitly designed into interfaces, because these assumptions critically shape their usability and accessibility (borrowing these terms from user interface design). *Usability* refers to the ease of use of a given interface and how easy it is for users to accomplish their goals; *accessibility* measures the extent to which it is possible to access all relevant information through a given interface when accessing it using various access methods.

In the context of designing Web service APIs, there is an ongoing debate about function-orientation, often associated with the concept of Remote Procedure Calls (RPC), vs. resource-orientation as the preferred architectural style for designing loosely coupled information systems. Function-orientation overlaps in many ways with the concepts of object-orientation (encapsulation and access through public methods), whereas resource-orientation focuses on the exchange of purposeful and self-contained documents. While the function-oriented style uses encapsulation and remote access to encapsulated data models through application-specific functions, the resource-oriented

^b This of course may be different if single entities or coordinated groups like standards organizations define shared data models, but even then this "one true model" only applies to the members of this group, and may not necessarily fit the needs and constraints of other potential users outside of the group.

style focuses on the exchange of openly exposed data models, and provides only a small and fixed number of access methods.

Metamodels for RPC and REST

This article focuses on resource-orientation as the underlying architectural style of an IT architecture. For function-orientation, the design of APIs usually follows similar guidelines as API design in system programming;³ these APIs are function-oriented and provide access to the entities that are locally implemented by the API provider. The “data model” of these APIs usually is rather simple; it is a set of simple types that can be used as types of input and output parameters of a function. The complexity of such an API thus lies in a concrete API design itself; that is, in the data model of the objects providing the functions, not in the metamodel of the API function definitions.

In contrast, with resource orientation, the situation is reversed. In this approach, the main focus is on document exchange, and the structure of these documents is usually much more elaborate than a function signature. On the surface, this difference typically manifests itself by a large number of functions with relatively simple signatures in function-oriented interfaces. On the other hand, resource-oriented interfaces typically define far fewer interactions, but they are coarse-grained ones that have more elaborate signatures in the form of complex resource representations.

Resource-orientation thus often needs more elaborate metamodels to define these resource representations. The selection of the metamodel is an important decision in the context of resource-orientation. Depending on the application, document design can come up with very different requirements for the data model; the spectrum of document models that are appropriate for application scenarios can range from simply structured regular sets of name/value pairs, to highly structured narrative document types mixing natural language with typed data, with a continuum of intermediary and mixed forms in between. The levels of abstraction and granularity, the robustness of the semantic

specification, and the extent to which standard conceptual components are reused in this data model are critical document design matters,² but they are not the central concerns of this article and will not be discussed further.

Our first important observation is that the selection of an API metamodel is of greater importance in resource-orientation than it is in function-orientation. Metamodels for resource-orientation typically are more complex, or at least span a larger range of possible requirements for concrete application scenarios, than in the case of function-orientation.

Another important observation is that on the Web, there often is no one true data model for a given application scenario. Instead, peers exchange representations of resources, which is the reason why the architectural style underlying the Web is called *Representational State Transfer (REST)*.¹ In REST, when two peers communicate, three models are involved: the inter-

nal model of one peer, the representation model that is used for communications, and the internal model of the other peer. It may be the case that both peers use the same internal model, but this is not a requirement of the Web. As long as peers share a common understanding of the intermediary representation model, they can interact.

One of the fundamental differences between function-orientation and REST is that REST’s goal is not integration, but collaboration. Function-orientation provides function-based access to a model that is implemented on the provider side; this model often is not exposed, and it is assumed that it is the model shared by all participants. On the other hand, REST provides an explicit representation of a model that is exchanged as a whole; and while participants are free to use that model for their implementation, they are also free to map it to a model they see as a better fit for their processing needs. When a model is

REST modeling layers.

shared, the participants can be more tightly coupled (often necessary for performance reasons) than when it is not, they can for example access the model through fine-grained functions that provide direct access to the model's components; in contrast, the loose coupling embodied in resource exchange enables easier substitution of service providers, which is often desirable for business reasons.

We call the internal models that participants have *system models*, and the representation model used for communication the *exchange model*. One complication in REST is the fact that both models have metamodels, and for successful collaboration it is necessary that the metamodel of the exchange model is made explicit, and is sufficiently described to match it to system models and thus to collaborate.

The Importance of Metamodels

Many real-life problems in Web Service architectures are rooted in implicit assumptions about metamodels. Two examples can be used to illustrate the role of the metamodel for the exchange model, and why it is important in the design of a REST architecture.

The first example starts with a tiny difference between “plain XML” and a data model's metamodel. If a data model is based on DTDs or XML Schema, it may use default values, which are not part of the XML syntax itself, but de-

finied by these schema languages for the metamodel. Applications processing the XML without using the DTD or the XML schema will not recognize these default values, because they are part of a specific schema-based metamodel, and encoded in the schema, not in the instance. Other XML mechanisms—for example XML Namespaces, XML Base, and XInclude—can also affect the interpretation of the model, because a model based on any of these specifications has to be interpreted according to their rules. Thus, even for something as interoperable as XML, it is essential to be explicit about the metamodel that is assumed as governing the data.

The second example is based on another popular metamodel, the Semantic Web. The Semantic Web's metamodel is RDF, and depending on the application, there often is a Schema defining a model for that metamodel (schemas for the Semantic Web are often called ontologies). RDF has several possible serializations; one of them is RDF/XML, where RDF triples are serialized as an XML document. For a peer in such a scenario to work correctly, it must parse the RDF/XML, and then transform the resulting XML document tree into a set of triples, a task that is non-trivial because of the syntax variations defined by RDF/XML. In addition, if there is a schema for the model, it must be taken into account when working with the RDF data, be-

cause Semantic Web applications should deal with the deductive closure of the RDF graph they process, not the (syntactic) graph itself. So while XML is being used in this scenario, it only appears on the level of the physical model (as the serialization format), whereas the logical model is RDF.^c

These two examples illustrate the importance of metamodels for the exchange model. Depending on the choice of the metamodel for the exchange model, peers are required to understand and implement the metamodel and the schema language that encodes the semantics as well as additional constraints for the data model.

There are a number of XML metamodels (after all, XML itself is only a syntax⁵), all of which are tree-based, but they use slight variations of the basic XML structures: The *XML Information Set (Infoset)* is the oldest metamodel of XML; its main contribution is that it includes *XML Namespaces* in the metamodel. The *Post Schema Validation Infoset (PSVI)* is XML Schema's variant of an XML metamodel; its main contribution is its support for typed trees. The *XQuery 1.0 and XPath 2.0 Data Model (XDM)* is becoming the defacto standard for XML metamodels; its main contribution is its support of sequences and thus the ability to represent non-XML values as well as XML trees. In addition to these basic XML metamodels, applications may choose to support additional metamodel features such as XInclude. Extensibility and versioning support often are desirable properties of data models, but for the XML metamodels listed here these are not readily available as metamodel features. Instead, they must be implemented by the data model, and best practices and design patterns can be used for guiding such a design.

APIs use Metamodels

When API designers use popular

^c Because RDF/XML is just a syntax for RDF data, it is also possible to serialize RDF as non-XML data; a popular format for this is *Notation 3 (N3)*. In such a scenario, it becomes even more apparent that RDF and XML have very little in common, and that RDF's XML syntax is just an implementation detail of a specific serialization, and does not provide interoperability on a higher level.

metamodels like RDF or UML, it is tempting to assume that users of the API will use the same metamodel as the provider. As a result, these API designers use their system metamodel as the exchange metamodel. We have identified this tendency to use system models as exchange models in Web contexts as *Web blindness*.⁵ Instead of creating an architecture based on the smallest possible set of assumptions (which in the case of Web data and services should be XML metamodels), the exchange metamodel then is dictated by the environment of the data or service provider.

We argue it is better practice in API design to think of an API as being designed by its users, rather than its providers, a point that also has been made for function-oriented APIs.³ Such a consumer-oriented definition of the exchange model focuses on the simplest possible set of assumptions about the consumers, making the API as usable and accessible as possible.

Metamodels in Practice

The following example illustrates how these metamodel considerations can inform the design of a resource-oriented API to use the smallest possible set of assumptions.

Let's assume that in a procurement application based on an ER system model, a purchase order is assembled to be sent to a supplier. The exchange model is defined as an XML Schema with additional semantic annotations, so that the application semantics (such as billing address and shipping address) are defined in the model. The order can then be assembled by matching the ER concepts of addresses to the corresponding XML concepts of the order document, that is, by mapping addresses in the system model to addresses in the exchange model. The order is then serialized as an XML document and sent to the supplier through some communications channel.

On the supplier side, when the order is received, the XML document is parsed and the exchange model is reconstructed so that it can be used by the supplier's order management application. Just to illustrate the point that the system models on the two sides can be radically different,

The argument that 'we use XML, which is a widely accepted data format, and thus our interface is easy to use' makes about as much sense as 'we use bits, which are a widely accepted data format, and thus our interface is easy to use.'

let's make the unlikely assumption that the supplier uses Semantic Web technologies. Using a mapping from XML to RDF, the exchange data can be mapped to RDF triples representing the order, because a matching process was used to identify how concepts in the exchange model can be mapped to the receiver's system model.

In such a scenario, the order is represented in three different conceptual models on its way from the sender to the receiver. This process is shown in the accompanying figure and described here in more detail.

Model Matching and Mapping

The existence of system and exchange models makes it necessary to match and map models. *Model matching* is the process of finding identical concepts in two models.^d In most cases, this process has to be done manually. If both models use the same metamodel, it often is possible to use tools to identify matching concepts, and even to generate code that will map data from one model to the other. For example, a popular functionality in XML editors is schema matching, where candidate conceptual matches between two schemas are made based on element and attribute names, and then refined manually in the editor if these matches are not exact.

It is important to point out that model matching usually is only partial. As there is no "one true model" of the domain, the system models of the participants typically cover larger application areas than the exchange model. It is also possible that the exchange model defines concepts that are not used in one of the system models, in which case this part of the exchange model will be ignored in the respective mapping. In this case, it is important which parts of the exchange model are mandatory and which are optional.

Model mapping is the process that takes data represented in one model, uses the conceptual matching of two models, and maps this data to another model. Mapping can take

^d This conveniently ignores the fact that in many cases concepts in different models are never fully identical; they often are partial and/or conditional matches only.

place between two models using the same metamodel (such as mapping one XML document to a semantically equivalent XML document using a different vocabulary), or it can take place between models based on different metamodels (such as mapping an XML document to RDF data). Mapping always takes place between a system model and an exchange model; this decouples all participating system models.

One of the biggest problems in the current stack of XML technologies is there (quite surprisingly) is no appropriate conceptual modeling language for XML. In terms of the stack of conceptual, logical, and physical models, XML Schema is not really a conceptual modeling language; it is primarily focused on markup design. It has some rudimentary conceptual modeling features, but is essentially a language for describing logical models. It is possible to use existing conceptual modeling languages (such as ER or UML) and automatically generate XML logical models (such as XML Schemas) from these, but these generated schemas often are not well-designed from a markup perspective. And critically, these generated schemas will not support any of the XML features that are not covered at all by the conceptual modeling language used as the starting point (most notably, support for narrative structures with mixed content).

This means the ideal process shown in the figure currently lacks support for the conceptual model layer of the exchange model (assuming XML as the exchange model's physical model). In many cases, applications use their own adhoc conceptual models and the logical model (such as an XML Schema) is then augmented with additional information to also serve as the conceptual model, or to refer to it.

Popular Web Metamodels

XML metamodels (as described earlier) are the most popular metamodels for Web data and services, but they sometimes are thought of as being too expensive in terms of processing, especially if the data model being represented is simple. An alternative physical model for XML metamodels is the *JavaScript Object Notation (JSON)*,

We strongly recommend the design of documents be informed by potential consumers and their use cases, not by the producers of documents.

which can only encode a subset of possible XML structures, and represents this subset in a way that can be interpreted as a JavaScript object. JSON is useful if the peers (or at least one of them) are JavaScript-based, and if the restrictions of the JSON syntax (no mixed content, restricted names, no difference between elements and attributes, no ordering) are acceptable for the exchange model.

A popular metamodel alternative to XML is RDF, the model of the Semantic Web. It is a metamodel based on triples, which can be used to construct complex graph structures. RDF works well for representing graphs, but has no specific support for the narrative end of the document type spectrum, which is well supported by XML. And while RDF is the universally accepted metamodel in the Semantic Web community, its use as an exchange model still limits the potential set of peers considerably more than an XML metamodel, because RDF tools are not as ubiquitous as XML tools.

Topology Troubles

The idealized matching of models shown in the figure can become complicated when the structures supported by the metamodels do not match very well. A typical example is when the data model is defined in UML and is able to represent arbitrary graphs. XML is tree-structured and thus not a good fit for such a model. In such a case, there are two basic alternatives:

- If the model is defined using some tool, in many cases these tools provide “export” or “generate schema” facilities for exporting the model as an XML Schema. While technically these generated schemas are XML representations of the model, typically they are rather cumbersome to work with on the XML level, because they are generated from a generic mapping of the metamodel to XML, and consequently do not take into account any specifics of the actual model.^e

- The other possibility is to use the approach we described earlier and as-

^e More advanced tools support annotating the model so that the mapping can be guided by annotations. This method yields better results as generated schemas, but the fundamental problem of misaligned metamodels of course cannot be changed.

sume a consumer of the model who does not know the non-XML metamodel or have tools to work with it. What kind of XML model would best serve the needs of that consumer? What are the use cases that should be served well by the XML model? After designing the XML model with the consumer in mind, a mapping between the two models can be defined.

This second alternative is more complex and thus expensive, because it requires a more specific approach, whereas the first one is based on a generic mapping. The second approach is more likely to produce a document design that will be usable and accessible on the XML level. If there is a realistic set of use cases to work with, the second approach can be based on real-world requirements.

The generic mapping approach often only gives lip service to the requirement to provide information in XML: While the generated XML model does represent the original data model, it often does so in ways that are only sufficiently usable and accessible for consumers using the non-XML metamodel. In such a scenario, the implicit assumption is that all consumers will use the non-XML metamodel, which restricts the set of potential users.

Such a restriction may be appropriate if it is a conscious decision. For example, if the metamodel is much more expressive than XML for the specific requirements of an application, it may be almost impossible to come up with a reasonable mapping to an XML model. However, if such a decision is made, there is no specific reason to use XML as the exchange model anymore, because it is treated as an opaque representation of a non-XML conceptual model, and only is used for the physical layer of the exchange model.

Physical XML

Using the figure as the explanation for the potential problems discussed in the previous section, it is easy to explain what happens. If the designer of an exchange format uses a non-XML conceptual metamodel because it seems to be a better fit for the data model, XML is only used as the physical layer for the exchange model. The logical layer in this case defines the

mapping between the non-XML conceptual model, and any reconstruction of the exchange model data requires the consumer to be fully aware of this mapping. In such a case, it is good practice to make users of the API aware of the fact that it is using a non-XML metamodel. Otherwise they might be tempted to base their implementation on a too small set of examples, creating implementations that are brittle and will fail at some point in time.

Real-world examples for this case are interfaces using RDF/XML. Naive implementations might assume that the data is encoded in a specific tree structure that can be accessed using XML tools such as XPath or XQuery. They might base their assumption on a set of sample documents, all of which might have been produced by the same RDF software and thus represent the same RDF/XML serialization strategy. However, such an implementation will most likely fail if it encounters documents where other serializations of the same RDF data are used. Thus, the only robust way in such a scenario is to use a full RDF/XML parser, and to first reconstruct the RDF data before doing anything with it.

Document Design Matters

This article has extended API design matters into the world of resource-orientation and REST as its currently most popular exponent. While function-oriented API design matters are mostly about the design of a concrete API, document design matters in the world of resource-orientation have a quite different emphasis. This emphasis is mostly on the models governing the design of an actual API; that is, the metamodels for these APIs. The design of REST APIs has more facets than only the selection of a particular metamodel for the representation of resources, but this is the most fundamental choice, and it directly affects the usability and accessibility of such an API.

We strongly recommend that the design of documents be informed by potential consumers and their use cases (such as, the environments in which these consumers are expected to develop and execute code, and the tasks they are expected to do), not by the producers of documents. By us-

ing the complete spectrum of possible consumers as input to the document design process for exchange models, it is possible to escape the implicit assumptions about supported metamodels and available tools that sometimes make data and services on the Web more difficult to work with than necessary.

Usability and accessibility of APIs, in particular of Web service APIs, should become central goals of API design, and a focus on well-designed document models based on widely available metamodels is a good starting point for these goals. While XML provides a proven foundation for defining exchange models, the lack of a conceptual model for XML makes it somewhat harder than it should be to define such a model. Because of this situation, using some non-XML conceptual model is a solution that is frequently chosen by practitioners, often resulting in exchange models embodying implicit assumptions. We hope this critical gap for XML-based resource orientation will soon be filled by some yet-to-be-invented language, capable of representing conceptual models for XML. Such a language would make it possible to better describe exchange models for resource-oriented APIs by supporting an easy way of generating schemas (logical models) for defining the representation (the markup) of exchange models. □

References

1. Fielding, R.T., Taylor, R.N. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology* 2, 2 (May 2002), 115–150.
2. Glushko, R.J., McGrath, T. *Document Engineering*. MIT Press, Cambridge, MA Aug. 2005.
3. Henning, M. API design matters. *ACM Queue* 5, 4 (May 2007), 24–36.
4. Prescod, P. Roots of the REST/SOAP debate. In *Proceedings of Extreme Markup Languages Conference* (Aug. 2002).
5. Wilde, E., Glushko, R.J. XML fever. *Commun. ACM* 51, 7 (July 2008), 26–31.

Erik Wilde (dret@berkeley.edu) is a visiting assistant professor in the School of Information at the University of California at Berkeley, where he is also technical director of the Information and Service Design program.

Robert J. Glushko (Glushko@ischool.berkeley.edu) has 20 years of experience with SGML and XML and is an adjunct professor at the University of California at Berkeley's School of Information. His teaching and research interests in business architecture and information systems and service design are complementary to and "higher in the stack" than Erik Wilde's.