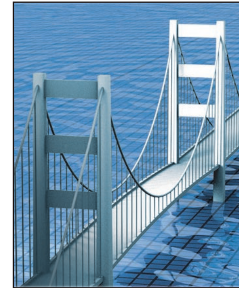
Demystifying RESTful Data Coupling



Steve Vinoski • *Verivue*

Last time, I explored how one of the constraints designed into the Representational State Transfer (REST) architectural style — the uniform interface — can increase chances for resource and service reuse. Compared to approaches such as Web services and the Web Services Description Language (WSDL), which promote specialization for each service interface, the uniform-interface constraint reduces client-server coupling and helps minimize gratuitous differences in interface and method semantics across disparate resources. REST isn't a silver bullet, but its flexibility and relative simplicity make it highly applicable not only to Web-scale systems but also to a wide variety of enterprise integration problems.

Developers who favor technologies that promote interface specialization typically raise two specific objections to the REST uniform-interface constraint. One is that different resources should each have specific interfaces and methods that more accurately reflect their precise functionality. This is rooted in the fact that most programming languages (especially those that are object-oriented) promote the development of specific interfaces, procedures, and methods for different software artifacts. This notion is so ingrained in many developers' minds that they consider it counterintuitive to apply a uniform general-purpose interface to anything — even to the heterogeneous services and resources found in a typical distributed system. Yet, those who raise this objection fail to properly consider the effects of networking and distribution. The REST architectural style specifically imposes constraints that help address problems typical of distributed systems, such as latency and state management. Among other things, the uniform-interface constraint helps enable visibility into client-server interactions, making it easier for developers to apply critical distributed systems concepts such as proxying, caching, intermediation, and monitoring.

The other objection to the concept of a uniform interface is that it merely shifts all coupling issues and other problems to the data exchanged between client and server. Detractors claim that, because of this shift, the uniform interface yields no overall benefits. I don't believe that's correct because it's based on the invalid assumption that the data exchanged in a REST system is just like the data exchanged in systems such as Web services and Corba, which require interface specialization. As we'll see, the purpose and form of exchanged data in REST differ significantly from those of other systems.

Specialized Data Issues

Service developers use languages such as the Corba Interface Definition Language and WSDL to define not only specialized service interfaces and methods but also the specialized data types passed through them. This isn't surprising: because these languages are strongly based on Remote Procedure Call (RPC), the definitions they encourage greatly resemble typical programming language method definitions. This resemblance means that such interface definition languages usually include constructs for defining structured types, which are similar in concept to C structs — essentially, ordered groupings of other data types, including other constructed types, with a theoretically unlimited level of possible nestings. This implies that such types can be arbitrarily complex. A common example of a simple specialized constructed type is an `Employee` type, which typically

1089-7801/08/$25.00 © 2008 IEEE

describes an employee's name, location, phone number, ID, and other relevant information.

In local applications written in typical enterprise languages such as Java and C++, defining specialized constructed types is not only normal, it's necessary. Without such types, passing groupings of related heterogeneous data to functions within the application would be overly complicated, perhaps prohibitively so. Keep in mind that constructed types aren't limited to pure data; Java and C++ classes are also constructed types, despite the fact that their data fields are usually private and accessed only through public methods.

Consider the coupling that these constructed types induce within a local application between a caller and a function or method it calls. Assuming a language like Java or C++, both the caller and the called method are compiled against the same definition of the constructed types they share. The caller and called method are also either compiled together into the same application or are strongly connected via dynamic loading. Either way, changing the constructed type's definition means recompiling both the caller and the called method. This ensures that both understand the same "shape" of the constructed type so that they can operate on it properly.

For local applications, this high level of coupling around constructed data types isn't much of a problem. Normally, the application's build system ensures that all affected files, modules, and packages are recompiled if a developer changes a type's definition. When a new version of the application is released, it's all built together and released as a whole, thus ensuring consistency of the constructed type definitions used across the system. I'm sure, though, that many of us have experienced the mysterious errors that result from missed recompilations, in which two or more linked pieces of code have different understandings of a constructed type's layout. Such errors are sometimes pretty hard to debug.

The coupling story around constructed types for distributed applications is, however, quite different: the larger the application's scale, the less likely that the individual applications involved were developed by the same team of developers or even by different teams in the same location. Moreover, there's no guarantee that the individual applications were developed at the same time, released on the same schedule, built on the same versions of the underlying software, or written in the same programming language. Yet, when the individual applications share an understanding of one or more specialized constructed data types, their level of coupling increases, and their independence is reduced. Depending on how such specialized data types are managed over time in terms of changes and versioning, their very existence in some distributed applications can be enough to prevent them from being used by any developer who isn't directly connected to the development team that manages the data types.

Still another problem with specialized constructed types in distributed systems is, ironically, that the types themselves are defined independently. We've all heard stories about or experienced firsthand the problems that can arise when independent applications developed within two different enterprises must be integrated because of mergers or acquisitions. Each application might have a different `Employee` definition, for example, thus forcing interactions involving employee data between the applications to rely on data transformation as the data passes through; the alternative is to rework the applications so that they share a specialized `Employee` definition.

If a distributed system comprises many independently developed applications, changing the definition of any specialized data type shared across those applications is fraught with problems. It ultimately boils down to independence versus centralized control. If you want your service to be useful for independently developed applications — whether you're working at Web scale or just within an enterprise — maintaining central control and coordination over your specialized constructed data type definitions for all applications that use them might be impossible.

Four Interface Constraints

The REST uniform-interface constraint isn't limited only to a set of operations such as HTTP's verb set. Four other key constraints support this constraint:[1]

1. resource identification;
2. resource manipulation through representations;
3. self-descriptive messages; and
4. hypermedia as the engine of application state.

Uniform resource identifiers (URIs) satisfy the requirements for constraint 1, so we won't focus on it other than to say that URIs are also critical for constraint 4. The other three constraints deserve more explanation.

For constraint 2, the name "representational state transfer" refers specifically to the fact that RESTful clients and servers interact by exchanging resource state representations. For example, a Web client performing an HTTP `GET` on a dynamic Web resource representing an employee might receive the current state of the employee details resource in the form of an HTML document. A client might also replace the state of the employee details by using `PUT` to send a new state representation, also in HTML form.

Several aspects of constraint 2 are related to data coupling between client and server. First, sending re-

source representations over the network seems to fly in the face of accepted principles of information hiding because it appears to require us to expose internal resource details. Fortunately, this isn't the case; a representation needn't reveal any details of the resource's implementation — for example, when you fetch an HTML Web page, you might get an HTML document read directly from the server's disk, or you might get the result from a program that constructed the HTML dynamically, perhaps by retrieving data from another service or a database. From a pure data-coupling perspective, exchanging resource representations in this fashion is no better or worse than data exchange in Web services or Corba.

Resource representations do, however, help alleviate some data-coupling problems because they're not tied to the underlying protocol. In REST, exchanged data are described using media types, and any given resource is free to represent its state using its choice of one or more media types. An HTTP client indicates desired representation formats by sending an `Accept` header, and a resource indicates the representation format of its response with a `Content-type` header. Contrast this data format flexibility with Corba, for example, in which exchanged data are defined in IDL, and marshaled in the standard Common Data Representation (CDR) format of the Object Management Group's standard Internet Inter-ORB Protocol (IIOP). For Web services and SOAP, data are normally exchanged in XML format. A REST resource's ability to handle state representations in different formats makes it easier to support a wider variety of client applications. It lets clients minimize their data coupling to the resource by choosing which representation is best for them.

Constraint 3 simply states that resource representations are self-describing, but this seemingly simple concept is quite important in REST. Because resource representation formats can vary, messages must indicate what format they carry. In HTTP, message payloads are identified using standard Internet Assigned Numbers Authority (IANA) Multipurpose Internet Mail Extensions (MIME) media types. HTML messages, for example, typically have the `text/html` MIME type, and JavaScript object notation (JSON) messages have the `application/json` MIME type. Developers are finding Atom-related MIME types, such as `application/atomsvc+xml`, particularly appealing for use in a wide variety of Web applications because they support resource syndication, publication, and editing (see RFCs 4287 and 5023 for more details[2,3]). For lists of all the registered MIME media types, visit the IANA site (www.iana.org/assignments/media-types/).

The fact that IANA MIME media types are globally standard data definitions helps reduce client-server data coupling. Independent parties can retrieve from the IANA site the details of any MIME media type they wish to support, and they can implement support for it in any language. As a result, numerous libraries for handling MIME types already exist for various programming languages. Rather than having to build specialized ad hoc code into your application to handle application-specific WSDL or IDL data types, your RESTful applications can reuse the appropriate libraries to handle the MIME types they work with. The fact that IANA controls these media type definitions means that they'll never change, which eliminates a lot of versioning-related churn and uncertainty.

Constraint 4 relates not so much to representation format as to representation content. In RESTful systems, applications interact with one or more resources — they, rather than the resources, maintain application state. Resource representations contain hyperlinks to help applications know how to perform application state transitions. For example, a resource designating a list of employees might return a representation containing a list of hyperlinks, each referring to a separate resource for each employee in the list. An application looking for information about a given employee need only follow the relevant hyperlinks using data and metadata within the representation as a guide. This approach helps with the client-server coupling problem by making application state transitions explicit within state representations, rather than implicitly hiding them behind and within collections of interface-specific methods. Keeping application state in the client rather than on the server can also help significantly with server scalability.

This approach helps with the coupling problem by making application state transitions explicit within state representations, rather than implicitly hiding them.

No Panacea

Is REST's approach to dealing with data coupling some sort of magic? Of course not. Although the REST constraints explored here can definitely help reduce data coupling when compared to

interface specialization approaches, there are still issues to watch for. For instance, sending representations typically means sending more data with each call than in RPC-oriented systems. Even though RESTful systems are often simpler and more efficient than their non-REST counterparts, this extra data overhead can sometimes cause efficiency problems. Another issue is that MIME types aren't a panacea. Consider the MIME type `application/xml`, used to indicate that a payload is some sort of XML document. Because of XML's extensibility, even if an application understands this MIME type, there's no guarantee that it will fully understand the XML payload. Still another issue is that there's no guarantee that a MIME type even exists for the representations you want to send or receive. Anyone is free to define a MIME type and register it, of course, but this takes time and effort that many people are unwilling or unable to expend. Furthermore, MIME types aren't the only game in town; some Web application developers prefer to define their data primarily in HTML or XHTML for browser viewing, but intersperse it with microformats (www.microformats.org) that allow applications to more easily interpret the data as well.

Ultimately, reducing interface and data coupling for your distributed applications isn't easy. RPC-oriented technologies such as Web services are intended primarily to extend programming language idioms and patterns across the network. By doing so, they hope to make the developer's job easier; unfortunately, this comes at the cost of significantly reduced scale, greater client-server coupling, and more difficult system modification and maintenance. The REST architectural style, on the other hand, makes very specific and highly useful trade-offs meticulously chosen to enhance the scalability, extensibility, manageability, and maintainability of distributed systems and applications. Yet, in my experience, applying REST actually requires less of a development effort. If you're building distributed applications, I firmly believe that by studying REST and adopting it wherever appropriate, you can significantly improve not only the applications you develop but also your own distributed system and integration development skills.

References

1. R.T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*, doctoral dissertation, Dept. of Computer Science, Univ. of Calif., Irvine, 2000.
2. M. Nottingham, *The Atom Syndication Format*, IETF recommendation, Dec. 2005; www.ietf.org/rfc/rfc4287.txt.
3. J. Gregorio and B. de hÓra, *The Atom Publishing Protocol*, IETF recommendation, Oct. 2007; www.ietf.org/rfc/rfc5023.

Steve Vinoski is a member of the technical staff at Verivue. He is a senior member of the IEEE and a member of the ACM. You can read his blog at http://steve.vinoski.net/blog/ and contact him at vinoski@ieee.org.