Tracking change and innovation in the enterprise software development community

InfoQ Explores

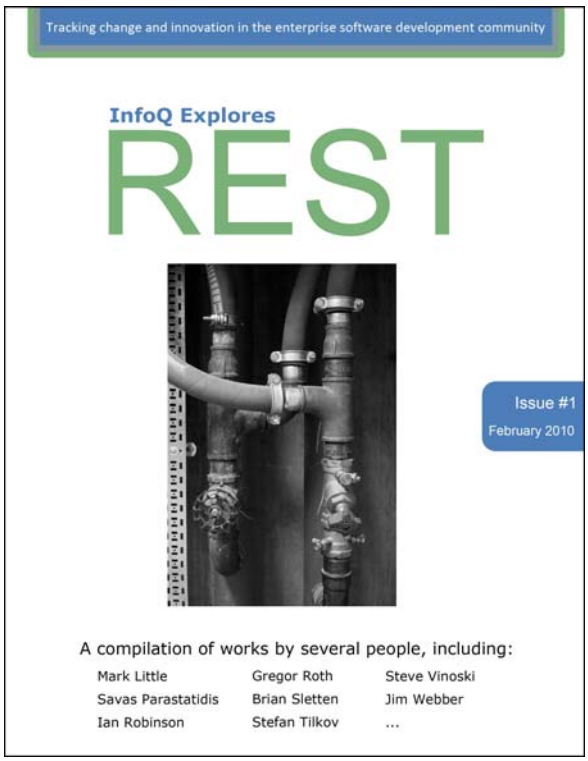
REST

Issue #1

March 2010

A compilation of works by several people, including:

Mark Little
Savas Parastatidis
Ian Robinson
Gregor Roth
Brian Sletten
Stefan Tilkov
Steve Vinoski
Jim Webber
...



InfoQ Explores: **REST**

Issue #1, March 2010

Chief Editor: Ryan Slobojan

Editors: Floyd Marinescu, Kevin Huo, Liu Shen

Feedback: ryan@infoq.com

Submit Articles: editors@infoq.com

InfoQueue
Enterprise Software Development Community

Content

[Articles]

[Interviews]

Articles

A Brief Introduction to REST

Author: Stefan Tilkov

You may or may not be aware that there is debate going on about the "right" way to implement heterogeneous application-to-application communication: While the current mainstream clearly focuses on web services based on SOAP, WSDL and the WS-* specification universe, a small, but very vocal minority claims there's a better way: REST, short for REpresentational State Transfer. In this article, I will try to provide a pragmatic introduction to REST and RESTful HTTP application integration without digressing into this debate. I will go into more detail while explaining those aspects that, in my experience, cause the most discussion when someone is exposed to this approach for the first time.

Key REST principles

Most introductions to REST start with the formal definition and background. I'll defer this for a while and provide a simplified, pragmatic definition: REST is a set of principles that define how Web standards, such as HTTP and URIs, are supposed to be used (which often differs quite a bit from what many people actually do). The promise is that if you adhere to REST principles while designing your application, you will end up with a system that exploits the Web's architecture to your benefit. In summary, the five key principles are:

- Give every "thing" an ID
- Link things together
- Use standard methods
- Resources with multiple representations
- Communicate statelessly

Let's take a closer look at each of these principles.

Give every "thing" an ID

I'm using the term "thing" here instead of the formally correct "resource" because this is such a simple principle that it shouldn't be hidden behind terminology. If you think about the systems that people build, there is usually a set of key abstractions that merit being identified. Everything that

should be identifiable should obviously get an ID — on the Web, there is a unified concept for IDs: The URI. URIs make up a global namespace, and using URIs to identify your key resources means they get a unique, global ID.

The main benefit of a consistent naming scheme for things is that you don't have to come up with your own scheme — you can rely on one that has already been defined, works pretty well on global scale and is understood by practically anybody. If you consider an arbitrary high-level object within the last application you built (assuming it wasn't built in a RESTful way), it is quite likely that there are many use cases where you would have profited from this. If your application included a Customer abstraction, for instance, I'm reasonably sure that users would have liked to be able to send a link to a specific customer via email to a co-worker, create a bookmark for it in their browser, or even write it down on a piece of paper. To drive home this point: Imagine what an awfully horrid business decision it would be if an online store such as Amazon.com did not identify every one of its products with a unique ID (a URI).

When confronted with this idea, many people wonder whether this means they should expose their database entries (or their IDs) directly — and are often appalled by the mere idea, since years of object-oriented practice have told us to hide the persistence aspects as an implementation detail. But this is not a conflict at all: Usually, the things — the resources — that merit being identified with a URI are far more abstract than a database entry. For example, an Order resource might be composed of order items, an address and many other aspects that you might not want to expose as individually identifiable resources. Taking the idea of identifying everything that is worth being identified further leads to the creation of resources that you usually don't see in a typical application design: A process or process step, a sale, a negotiation, a request for a quote — these are all examples of "things" that merit identification. This, in turn, can lead to the creation of more persistent entities than in a non-RESTful design.

Here are some examples of URIs you might come up with:

```
http://example.com/customers/1234
http://example.com/orders/2007/10/776654
http://example.com/products/4554
http://example.com/processes/salary-increase-234
```

As I've chosen to create human-readable URIs — a useful concept, even though it's not a pre-requisite for a RESTful design — it should be quite easy to guess their meaning: They obviously identify individual "items". But take a look at these:

```
http://example.com/orders/2007/11
http://example.com/products?color=green
```

At first, these appear to be something different — after all, they are not identifying a thing, but a collection of things (assuming the first URI identifies all orders submitted in November 2007, and the second one the set of green products). But these *collections* are actually things — resources — themselves, and they definitely merit identification.

Note that the benefits of having a single, globally unified naming scheme apply both to the usage of

the Web in your browser and to machine-to-machine communication.

To summarize the first principle: Use URIs to identify everything that merits being identifiable, specifically, all of the "high-level" resources that your application provides, whether they represent individual items, collections of items, virtual and physical objects, or computation results.

Link things together

The next principle we're going to look at has a formal description that is a little intimidating: "Hypermedia as the engine of application state", sometimes abbreviated as HATEOAS. (Seriously — I'm not making this up.) At its core is the concept of *hypermedia*, or in other words: the idea of *links*. Links are something we're all familiar with from HTML, but they are in no way restricted to human consumption. Consider the following made-up XML fragment:

```
<order self='http://example.com/customers/1234' >
   <amount>23</amount>
   <product ref='http://example.com/products/4554' />
   <customer ref='http://example.com/customers/1234' />
</order>
```

If you look at the product and customer links in this document, you can easily imagine how an application that has retrieved it can "follow" the links to retrieve more information. Of course, this would be the case if there were a simple "id" attribute adhering to some application-specific naming scheme, too — *but only within the application's context*. The beauty of the link approach using URIs is that the links can point to resources that are provided by a different application, a different server, or even a different company on another continent — because the naming scheme is a global standard, all of the resources that make up the Web can be linked to each other.

There is an even more important aspect to the hypermedia principle — the "state" part of the application. In short, the fact that the server (or service provider, if you prefer) provides a set of links to the client (the service consumer) enables the client to move the application from one state to the next by following a link. We will look at the effects of this aspect in another article soon; for the moment, just keep in mind that links are an extremely useful way to make an application dynamic.

To summarize this principles: Use links to refer to identifiable things (resources) wherever possible. Hyperlinking is what makes the Web the Web.

Use standard methods

There was an implicit assumption in the discussion of the first two principles: that the consuming application can actually *do* something meaningful with the URIs. If you see a URI written on the side of a bus, you can enter it into your browser's address field and hit return — but how does your browser know what to do with the URI?

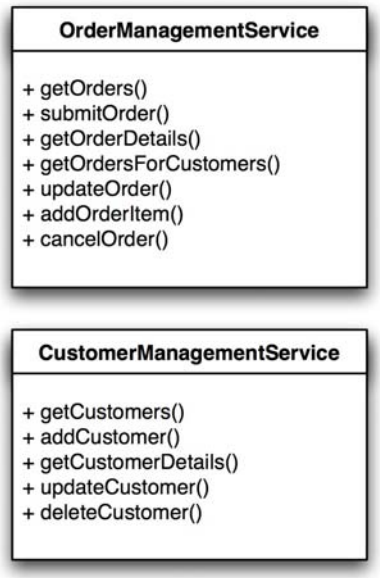
It knows what to do with it because every resource supports the same interface, the same set of methods (or operations, if you prefer). HTTP calls these *verbs*, and in addition to the two everyone knows (GET and POST), the set of standard methods includes PUT, DELETE, HEAD and OPTIONS. The meaning of these methods is defined in the HTTP specification, along with some guarantees about their behavior. If you are an OO developer, you can imagine that every resource in a RESTful HTTP scenario extends a class like this (in some Java/C#-style pseudo-syntax and concentrating on the key methods):

```
class Resource {
    Resource(URI u);
    Response get();
    Response post(Request r);
    Response put(Request r);
    Response delete();
}
```

Because the same interface is used for every resource, you can rely on being able to retrieve a *representation* — i.e., some rendering of it — using GET. Because GET's semantics are defined in the specification, you can be sure that you have no obligations when you call it — this is why the method is called "safe". GET supports very efficient and sophisticated caching, so in many cases, you don't even have to send a request to the server. You can also be sure that a GET is *idempotent* — if you issue a GET request and don't get a result, you might not know whether your request never reached its destination or the response got lost on its way back to you. The idempotence guarantee means you can simply issue the request again. Idempotence is also guaranteed for PUT (which basically means "update this resource with this data, or create it at this URI if it's not there already") and for DELETE (which you can simply try again and again until you get a result — deleting something that's not there is not a problem). POST, which usually means "create a new resource", can also be used to invoke arbitrary processing and thus is neither safe nor idempotent.

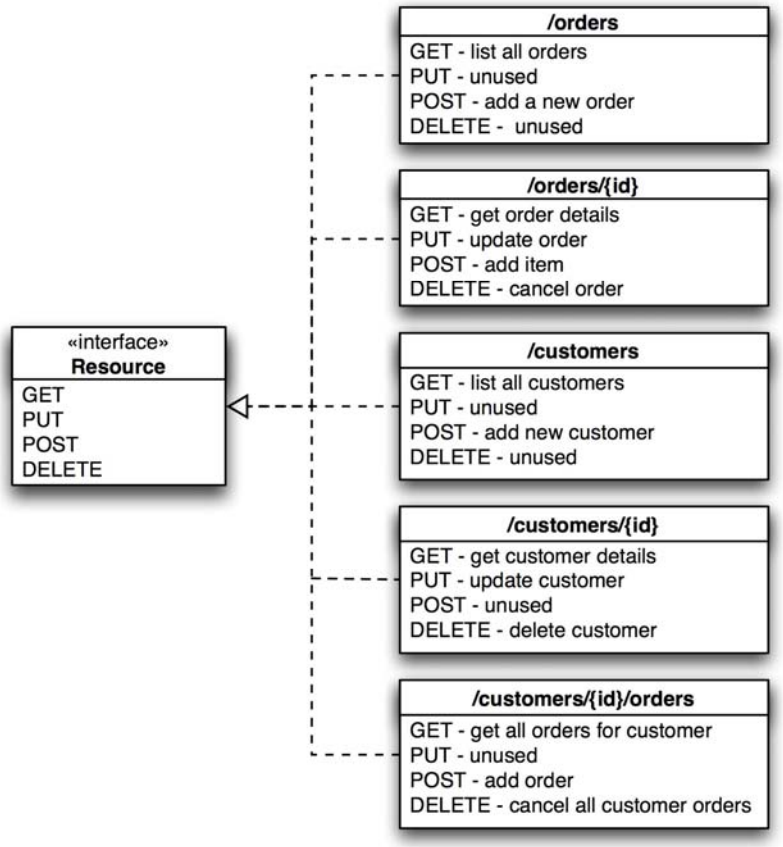
If you expose your application's functionality (or service's functionality, if you prefer) in a RESTful way, *this principle and its restrictions apply to you as well*. This is hard to accept if you're used to a different design approach — after all, you're quite likely convinced that *your* application has much more logic than what is expressible with a handful operations. Let me spend some time trying to convince you that this is not the case.

Consider the following example of a simple procurement scenario:

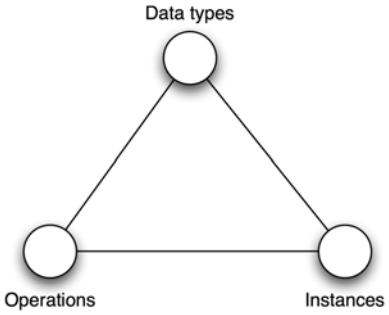


You can see that there are two services defined here (without implying any particular implementation technology). The interface to these services is specific to the task — it's an OrderManagement and CustomerManagement service we are talking about. If a client wants to consume these services, it needs to be coded against this particular interface — there is no way to use a client that was built before these interfaces were specified to meaningfully interact with them. The interfaces define the services' application protocol.

In a RESTful HTTP approach, you would have to get by with the generic interface that makes up the *HTTP application protocol*. You might come up with something like this:



You can see that what have been specific operations of a service have been mapped to the standard HTTP methods — and to disambiguate, I have created a whole universe of new resources. “That’s cheating!”, I hear you cry. No - it’s not. A GET on a URI that identifies a customer is just as meaningful as a getCustomerDetails operation. Some people have used a triangle to visualize this:



Imagine the three vertices as knobs that you can turn. You can see that in the first approach, you have many operations and many kinds of data and a fixed number of “instances” (essentially, as many as you have services). In the second, you have a fixed number of operations, many kinds of data and many objects to invoke those fixed methods upon. The point of this is to illustrate that you can basically express anything you like with both approaches.

Why is this important? Essentially, it makes your application part of the Web — its contribution to what has turned the Web into the most successful application of the Internet is proportional to the number of resources it adds to it. In a RESTful approach, an application might add a few million customer URIs to the Web; if it's designed the same way applications have been designed in CORBA times, its contribution usually is a single "endpoint" — comparable to a very small door that provides entry to a universe of resource only for those who have the key.

The uniform interface also enables every component that understands the HTTP application protocol to interact with your application. Examples of components that benefit from this are generic clients such as curl and wget, proxies, caches, HTTP servers, gateways, even Google/Yahoo!/MSN, and many more.

To summarize: For clients to be able to interact with your resources, they should implement the default application protocol (HTTP) correctly, i.e. make use of the standard methods GET, PUT, POST, DELETE.

Resources with multiple representations

We've ignored a slight complication so far: how does a client know how to deal with the data it retrieves, e.g. as a result of a GET or POST request? The approach taken by HTTP is to allow for a separation of concerns between handling the data and invoking operations. In other words, a client that knows how to handle a particular data format can interact with all resources that can provide a representation in this format. Let's illustrate this with an example again. Using HTTP content negotiation, a client can ask for a *representation* in a particular format:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml
```

The result might be some company-specific XML format that represents customer information. If the client sends a different request, e.g. one like this:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

The result could be the customer address in VCard format. (I have not shown the responses, which would contain metadata about the type of data in the HTTP Content-type header.) This illustrates why ideally, the representations of a resource should be in standard formats — if a client "knows" both the HTTP application protocol and a set of data formats, *it can interact with any RESTful HTTP application in the world* in a very meaningful way. Unfortunately, we don't have standard formats for everything, but you can probably imagine how one could create a smaller ecosystem within a company or a set of collaborating partners by relying on standard formats. Of course all of this does not only apply to the data sent from the server to the client, but also for the reverse direction — a server that can consume data in specific formats does not care about the particular type of client,

provided it follows the application protocol.

There is another significant benefit of having multiple representations of a resource in practice: If you provide both an HTML and an XML representation of your resources, they are consumable not only by your application, but also by every standard Web browser — in other words, information in your application becomes available to everyone who knows how to use the Web.

There is another way to exploit this: You can turn your application's Web UI into its Web API — after all, API design is often driven by the idea that everything that can be done via the UI should also be doable via the API. Conflating the two tasks into one is an amazingly useful way to get a better Web interface for both humans and other applications.

Summary: Provide multiple representations of resources for different needs.

Communicate statelessly

The last principle I want to address is *stateless communication*. First of all, it's important to stress that although REST includes the idea of statelessness, this does not mean that an application that exposes its functionally cannot have state — in fact, this would render the whole approach pretty useless in most scenarios. REST mandates that state be either turned into resource state, or kept on the client. In other words, a server should not have to retain some sort of communication state for any of the clients it communicates with beyond a single request. The most obvious reason for this is scalability — the number of clients interacting would seriously impact the server's footprint if it had to keep client state. (Note that this usually requires some re-design — you can't simply stick a URI to some session state and call it RESTful.)

But there are other aspects that might be much more important: The statelessness constraint isolates the client against changes on the server as it is not dependent on talking to the same server in two consecutive requests. A client could receive a document containing links from the server, and while it does some processing, the server could be shut down, its hard disk could be ripped out and be replaced, the software could be updated and restarted — and if the client follows one of the links it has received from the server, it won't notice.

REST in theory

I have a confession to make: What I explained is not really REST, and I might get flamed for simplifying things a little too much. But I wanted to start things a little differently than usual, so I did not provide the formal background and history of REST in the beginning. Let me try to address this, if somewhat briefly.

First of all, I've avoided taking great care to separate REST from HTTP itself and the use of HTTP in a RESTful way. To understand the relationship between these different aspects, we have to take a look at the history of REST.

The term REST was defined by Roy T. Fielding in his PhD thesis (you might actually want to follow that link — it's quite readable, for a dissertation at least). Roy had been one of the primary designer of many essential Web protocols, including HTTP and URIs, and he formalized a lot of the ideas behind them in the document. (The dissertation is considered "the REST bible", and rightfully so — after all, the author invented the term, so by definition, anything he wrote about it must be considered authorative.) In the dissertation, Roy first defines a methodology to talk about *architectural styles* — high-level, abstract patterns that express the core ideas behind an architectural approach. Each architectural style comes with a set of *constraints* that define it. Examples of architectural styles include the "null style" (which has no constrains at all), pipe and filter, client/server, distributed objects and — you guessed it — REST.

If all of this sounds quite abstract to you, you are right — REST in itself is a high-level style that could be implemented using many different technologies, and instantiated using different values for its abstract properties. For example, REST includes the concepts of resources and a uniform interface — i.e. the idea that every resource should respond to the same methods. But REST doesn't say which methods these should be, or how many of them there should be.

One "incarnation" of the REST style is HTTP (and a set of related set of standards, such as URIs), or slightly more abstractly: the Web's architecture itself. To continue the example from above, HTTP "instantiates" the REST uniform interface with a particular one, consisting of the HTTP verbs. As Fielding defined the REST style after the Web — or at least, most of it — was already "done", one might argue whether it's a 100% match. But in any case, the Web, HTTP and URIs are the only major, certainly the only relevant instance of the REST style as a whole. And as Roy Fielding is both the author of the REST dissertation and has been a strong influence on the Web architecture's design, this should not come as a surprise.

Finally, I've used the term "RESTful HTTP" from time to time, for a simple reason: Many applications that use HTTP don't follow the principles of REST — and with some justification, one can say that using HTTP without following the REST principles is equal to abusing HTTP. Of course this sounds a little zealous — and in fact there are often reasons why one would violate a REST constraint, simply because every constraint induces some trade-off that might not be acceptable in a particular situation. But often, REST constraints are violated due to a simple lack of understanding of their benefits. To provide one particularly nasty example: the use of HTTP GET to invoke operations such as deleting an object violates REST's safety constraint and plain common sense (the client cannot be held accountable, which is probably not what the server developer intended). But more on this, and other notable abuses, in a follow-up article.

Summary

In this article, I have attempted to provide a quick introduction into the concepts behind REST, the architecture of the Web. A RESTful HTTP approach to exposing functionality is different from RPC, Distributed Objects, and Web services; it takes some mind shift to really understand this difference. Being aware about REST principles is beneficial whether you are building applications that expose a

Web UI only or want to turn your application API into a good Web citizen.

Link: http://www.infoq.com/articles/rest-introduction

Related Contents：

- Interview with Guilherme Silveira, creator of Restfulie
- Is JAX-RS, or RESTeasy, un-RESTful?
- IBM WebSphere Embraces REST
- Business Case For REST
- Is CRUD Bad for REST?

Articles

Resource-Oriented Architecture: The Rest of REST

Author: Brian Sletten

Series Introduction

Think for a moment, if you can, back to a time before the Web. Imagine trying to explain the impending changes to your hapless contemporaries. It is likely they would simply not be able to fathom the impacts that the Web's emergence would have on nearly every aspect of their lives. In retrospect, it feels like a tsunami caught us off-guard and forever altered the landscape around us. The reality is more pedestrian, however. It was a deliberate series of technical choices that built upon each other that yielded the results we have experienced.

Now, pause and reflect upon the idea that you are probably in a similar position to those incredulous pre-Web types you were just trying to enlighten. Unless you have been paying close attention, you are about to be caught off-guard again as it feels like a new wave crashes upon our economic, social, technological and organizational landscapes. While the resulting changes will feel like they occur overnight, the reality is that they have been in the works for years and are just now producing tangible results. This new wave is about a Web that has evolved beyond documents into Webs of Data, both personal and private. We will no longer focus on information containers, but on information itself and how it is connected.

This wave has been in the works for years and is again being driven by the deliberate adoption of specific choices and technologies. These choices are combining to solve the problems caused by the inexorable march of technological change, business flux, new and varied data sources and the ubiquitous, expensive and failure-prone efforts that have cost millions and delivered insufficient value. Web Services and Service-Oriented Architectures (SOA) were supposed to be part of the answer, but the elegance of their visions have been forever stained by the inelegance of their technical solutions.

The beauty is that we are not starting from scratch. We are building upon the technology we have in place to grow these data webs organically. We can wrap our databases, libraries, services and other content sources with a new set of abstractions that will help us off the treadmill we have been on. We are integrating the public Web of Data with our own, privately held data. The incremental adoption of these technologies is yielding new capabilities that will, in turn, unlock further capabilities.

This is the first article in a new series to highlight the evolution of information-oriented systems that got us to where we are and provide a roadmap to where we are going. Despite what it may seem on the surface, these choices are neither ad hoc nor esoteric, but rather foundational decisions based on a long tradition of academia and applied engineering.

We will start by revisiting the REpresentational State Transfer (REST) architectural style. Oft quoted and even more often misunderstood, this manner of building networked software systems allows us to merge our documents, data and information-oriented services into a rich, logical ecosystem of named resources. From there, we will introduce the vision of the Semantic Web and walk through its core technologies represented by a flexible and extensible data model and the ability to query it. We will see how to incorporate relational data, content from documents, spreadsheets, RSS feeds, etc. into a rich web of reusable content.

After we present the basics, we will walk through a variety of successful efforts building on these technologies and then return to reclaiming the vision promised to us by proponents of Web Services technologies. We will describe a process where we can achieve something of a Unified Theory of Information Systems; one that not only handles, but embraces the kind of technical and social change that has been painful and intractable to manage in the past.

There has been too much hype surrounding the Semantic Web, but there have also been a steady stream of quiet successes. This series will be a pragmatic guide into both new and familiar territory. We will connect the technologies in deeper ways than perhaps you have seen before. We will highlight events and actions by companies, government organizations and standards bodies that indicate that this is happening and it will change everything. We will show how a very large difference in your system implementation can often be made through subtle shifts in perspective and adoption of standards that are designed to facilitate change.

The first step, is to embrace a common naming scheme for all aspects of our infrastructure. A Service-Only Architecture usually ignores the data that flows through it. At the end of the day, our organizations care about information first and foremost. REST and the Web Architecture puts this priority up front and lays the foundation for the remainder of our discussion.

The Rest of REST

It has become fashionable to talk about the REpresentational State Transfer (REST) as something of a weapon in the War On Complexity. The enemies in this war, according to some, are SOAP and the Web Services technology stack that surrounds it. This Us vs Them rhetoric brings passion to the table, but rarely meaningful dialogue so people remain confused as to the underlying message and why it is important. The goal is not to replace SOAP; the goal is to build better systems.

REST is not even a direct replacement for SOAP. It is not some kind of technology of convenience; a simple solution for invoking Web Services through URLs. The management of information resources is not the same thing as invoking arbitrary behavior. This confusion leads people to build "RESTful" solutions that are neither RESTful, nor good solutions.

REST derives its benefits as much from its restrictions as it does its resultant flexibility. If you read Dr. Roy Fielding's thesis (which you are encouraged to do), you will learn that the intent was to describe how the combination of specific architectural constraints yields a set of properties that we find desirable in networked software systems. The adoption of a uniform interface, the infamous Uniform Resource Locator (URL), contributes to the definition of REST, but is insufficient to define it. Likewise, interfaces that simply expose arbitrary services via URLs will not yield the same benefits we have seen so successfully in the explosion of the Web. It takes a richer series of interactions and system partitioning to get the full results.

Most people understand that REST involves requesting and supplying application state of information resources through URLs via a small number of verbs. You retrieve information by issuing GET requests to URLs, you create or update via POST and PUT, and remove information via DELETE requests.

This summary is not incorrect, but it leaves too much out. The omissions yield degrees of freedom that unfortunately often allow people to make the wrong decisions. In this gap, people create URLs out of verbs which eliminates the benefit of having names for "things". They think REST is just about CRUD operations. They create magical, unrelated URLs that you have to know up front how to parse, losing the discoverability of the hypertext engine. Perhaps most unforgivably, they create URLs tied solely to particular data formats, making premature decisions for clients about the shape of the information.

Understanding the full implications of REST will help you avoid these problems; it will help you to develop powerful, flexible and scalable systems. But it is also the beginning of a new understanding of information and how it is used. Upon this foundation of Web architecture, the application of the remaining technologies of the Semantic Web will yield unprecedented power in how we interact with each other as individuals, governments, organizations and beyond. This is why we begin with a deeper dive into the parts of REST that many people do not understand and therefore do not discuss. These topics include the implications of:

- URLs as identifiers
- Freedom of Form
- Logically-connected, Late-binding Systems
- Hypertext as the Engine of State Transfer (HATEOS)

URLs as Identifiers

We have already established that most people know about URLs and REST. It seems clear that they understand that a URL is used for invoking a service, but it is not clear that they get the larger sense of a URL as a name for information. Names are how we identify people, places, things and concepts. If we lack the ability to identify, we lack the ability to signify. Imagine Abbott and Costello's infamous "Who's on First?" skit on a daily basis. Having names gives us the ability to disambiguate and identify something we care about within a context. Having a name and a common context allows us to make

reference to named things out of that context.

The Uniform Resource Identifier (URI) is the parent scheme. It is a method for encoding other schemes depending on whether we want them to include resolution information or not. Librarians and other long-term data stewards like names that will not change. A Uniform Resource Name (URN) is a URI that has no location information in it; nothing but name is involved. The good news is that these names will never break. The bad news is that there is no resolution process for them. An example of a URN is an ISBN number for a book:

```
urn:isbn:0307346617
```

In order to find more information about this book, you would have to find a service that allows you to look up information based on the ISBN number.

If nothing about the context of our systems and information ever changed, we would probably always want to include resolution information in our resource names so we could resolve them. But anyone who has been handed a broken link knows we want longer-lived names for really important stuff. Looking at our history of using URLs, we have done some silly things when we created ones such as:

```
http://someserver.com/cgi-bin/foo/bar.pl
```

```
http://someserver.com/ActionServlet?blah=blah
```

```
http://someserver.com/foo/bar.php
```

The problem with these URLs is that the technology used to produce a result is irrelevant to the consumer of information. There is no good reason to create URLs like that. The focus should be on the information, not the technology. Implementation technologies change over time. If you abandon them, for instance, any system that has a link to the Perl, Servlet or PHP-based URL will break. We will address some infrastructure to solve this problem in future articles, for now, we will just try to make careful choices in the names we give our information resources.

Despite being fragile, the URL scheme does allow us to disambiguate information references in a global context.

```
http://company1.com/customer/123456
```

is distinct and distinguishable from

```
http://company2.com/customer/123456
```

in ways that a decontextualized identifier like '123456' is not.

To ground the concept into a larger information systems framework, you can think of a URL as a primary key that is not specific to a particular database. We can make references to an item via its URL in dozens of different databases, documents, applications, etc. and know that we are referring to the same thing because we have a unique name in a global context. We will use this property in future discussions to describe and connect RESTful systems to other content and metadata.

The next aspect of URLs that bears discussion is their universal applicability. We have a common naming scheme that allows us to identify:

- documents (reports, blogs, announcements)
- data (results, instance information, metadata)
- services (REST!)
- concepts (people, organizations, domain-specific terms)

We do not need to come up with a different mechanism to refer to each different category of things. A careful application of some specific guidelines allows us to blur the distinctions between these things which brings us to the last point for now about URLs. Not only are these names useful in order to refer to information we care about, but systems that receive these references can simply ask for them. The 'L' in URL (locator) gives us the capacity to resolve the thing, not knowing anything else about it. We can usually invoke the same basic operations on everything we can name. Issuing a GET request to a URL representing a document, some data, a service to produce that data or an abstract, non-network-addressable concept all work fundamentally the same way. For those things we have the permission to manipulate, we can also create, modify or delete them using similar means.

Freedom of Form

Our experience of the Web has been somewhat passive with respect to the shape of information. When we click on a link, we expect the content to come back in a particular form, usually HTML. That is fine for many types of information, but the architecture supports a much more conversational style allowing clients to request information in a preferred form.

To understand why this is useful, consider a company's sales report. It is easy to imagine this being useful to executives, sales people, other employees, clients and investors as an indication of how a company is performing. A possible name for such a report could include the year and the quarter in the URL:

```
http://company1.com/report/sales/2009/qtr/3
```

We might contrast this with a sales report for the month of March:

```
http://company1.com/report/sales/2009/month/3
```

Both are good, logical names that are unlikely to break over time. It is a compelling vision that people could simply type such a URL into a browser and get the information they seek rendered as HTML. The reports could be bookmarked, e-mailed, linked to, etc.; all the things we love about the Web.

The problem is that the information is locked into its rendered form (until we introduce technologies like GRDDL and RDFa later in this series!). We used to try to scrape content from pages, but gave up in disgust. As the page layout changes, our scripts break.

If you were a programmer for this company and wanted to get to the information directly, you might like to request it as XML. You could get back raw, structured data that you could validate against a schema. HTTP and REST make this trivial as long as the server knows how to respond. By passing in an "Accept: application/xml" header to your request, you could indicate a preference (or requirement) for XML. On success, you will get back a byte-stream with a MIME type indicating that your request has been honored. On failure, the server will indicate via a 406 Error that it cannot help you. In that case, you might want to contact the department responsible for this information and request they add the support you need; something they can do without breaking any existing clients. If you were a business analyst, you might think that XML has sharp points and can hurt you, so you might like to request it back as a spreadsheet, a format that is easily incorporated into your existing workflows, tools and processes.

The point is that the logical name for the report is easily converted into various forms at the point it is requested. It is equally easy to run systems that accept modifications back in the various forms. The client has no visibility into how the information is actually stored, they just know that it works for them. This freedom is wholly underused by people building RESTful systems. When they stand up a service and decide that they will only return XML, they miss the potential value REST has to an organization.

Because many developers are either unaware of content negotiation or find it difficult to test in a browser, they define different URLs for the different formats:

```
http://company1.com/report/sales/2009/qtr/3/report.html
```

```
http://company1.com/report/sales/2009/qtr/3/report.xml
```

```
http://company1.com/report/sales/2009/qtr/3/report.xls
```

This developer convenience becomes a limitation once you escape the confines of a particular use. In essence, we now have three information resources, not one that can be rendered in different forms. Not only does this fork the identity in the global context, it also prematurely commits other clients to a particular form. If you pass a reference to a URL as part of a workflow or orchestration you are robbing the upstream clients from the freedom to choose the form of the data.

There are several ways to test a proper RESTful service without using a browser, for example:

```
curl -H "Accept: application/xml" -O
http://company1.com/report/sales/2009/qtr/3
```

using the popular curl program. Any reasonable HTTP client will provide similar capabilities.

The benefits of supporting a rich ecosystem of negotiable data forms may not be immediately obvious, but once you wrap your head around it, you will see it as a linchpin toward long-lived, flexible systems that favor the client, not the developer.

Logically-Connected, Late-Binding Systems

Once you commit to good, logical names for your information resources, you will discover some additional benefits that fall out of these decisions. Named references can safely and efficiently be passed back as results without returning actual data. This has strong implications for large and sensitive data sets, but it also makes possible technical and architectural migration.

For the same reasons pointers are useful in languages like C and C++, URLs as references to data are more compact and efficient to hand off to potential consumers of information. Large data sets such as financial transactions, satellite imagery, etc. can be referenced in workflows without requiring all participants to suffer the burden of handling the large content volume.

Any orchestration that touches actual data must consider the security implications of passing it on to other systems. It quickly becomes untenable to provide perfect knowledge of who is allowed to do what at every step of a process. If a reference is passed from step to step, it is up to the information source to enforce access. Some steps may not require access to the sensitive information and could therefore be excluded from receiving it when they resolve the reference.

This means the late-binding resolution can factor in the full context of the request. A particular user accessing a resource from one application might have a business need to see sensitive information. The same person using a different application might not have a business justification to the same data. A RESTful service could inspect session tokens and the like to enforce this access policy declaratively. This level of specificity is required to prevent internal fraud, often the biggest risk in systems that deal with sensitive content. The details of such a system are going to be implementation-specific and are largely orthogonal to the process of naming and resolving logically-named content.

Dependency on a logical connection allows clients to be protected against implementation changes. When popular websites shift from one technology to another, they are usually successful at hiding these changes from their users. RESTful services do the same thing. This gives us the freedom to wrap legacy systems with logical interfaces and leave them in place until there is a business reason to invest in a new implementation. When that happens, clients can be protected from being affected.

In addition to mediating technology changes, RESTful systems allow you to embrace a variant of Postel's Law: Be Conservative in what you do; be Liberal in what you accept from others. You can maintain strict content validation of what you accept and return. However, if you have an existing client base that is providing you content in a given form, you are free to allow other clients to provide different forms, different schemas, etc. without affecting the existing clients. Systems that closely associate a contract with an endpoint tend not to have this freedom which makes them more brittle and quickly fragmented.

Hypertext As the Engine of State Transfer (HATEOS)

As systems come across references to information resources, many people think there needs to be some sort of description language to indicate what is possible or should be done with it. The reality is that a well-considered RESTful system usually does not require this concept. This is difficult for SOAP developers to accept, but it has to do with the constraints of the architectural style. Because we treat information resources as things to manipulate through a uniform interface (the URL!) and restrict our efforts to a small set of verbs, there really is no need to describe the service.

If you find yourself confused on this point, it is probably an architectural smell that you are conflating manipulating resources with invoking arbitrary behavior. The REST verbs provide the full set of operations to apply to an information resource. Certainly, you need to know what information is being returned so you know how to process it, but that is what MIME types are for. While it is usually preferable to reuse known types (application/xml, image/png, etc.), many developers do not realize that they can create their own application-specific data types if necessary.

In the larger arc of this article series, we will address the problems of finding and binding arbitrary resources using rich metadata. For now, we will simply keep in mind Roy's underscoring of the importance of "hypertext as the engine of state transfer" (obliquely referred to as "HATEOS" by RESTafarians). This is perhaps the most misunderstood portion of the thesis. To get its full implication, we need to revisit how the Web works.

You type a URL into the browser and it issues an HTTP GET request for that resource. Invariably, the server responds with a bytestream, a response code (usually 200 on success) and a MIME type indicating that the response is HTML. The browser decides it knows how to handle this type and parses the result into a document model of some sort. Within that model, it finds references to other resources: links, images, scripts, style sheets, etc. It treats each one differently, but it discovers them in the process of resolving the original resource. There is no service description; the browser, as a client, simply knows how to parse the result.

The same mechanism should be employed for RESTful services. The URLs themselves should not be "magical". A client should not be required to know how to parse a URL or have any special knowledge of what one level in the hierarchy means over another one. RESTful clients should retrieve a resource, investigate the returned MIME type and parse the result. As such, a client should know how to parse the returned type.

For example, a client might receive a reference to the main RESTful service for the reporting service we described above:

```
http://company1.com/report/
```

If requested from a browser, it could return an HTML document that has references to:

```
http://company1.com/report/sales
```

which the user could click through to find a list of years to browse. The point is that the browser has

no special knowledge of the URL structure, but it knows how to parse the result and present the content to the user in a way she can explore.

The same can be true of other MIME type responses. For example, requesting the 2009 quarterly reports as XML:

```
http://company1.com/reports/sales/2009/qtr
```

could yield:

```
<reports>
   <description>2009 Quarterly Reports</description>
   <report name="First Quarter"
src="http://company1.com/reports/sales/2009/qtr/1"/>
   <report name="Second Quarter"
src="http://company1.com/reports/sales/2009/qtr/2"/>

   <report name="Third Quarter"
src="http://company1.com/reports/sales/2009/qtr/3"/>
</reports>
```

You can think of the URL as a vector through an information space. Each level points you closer to the ultimate resource. Different paths can yield the same results. The client will have to know how to parse these results, but by giving the response an identifiable type, we can trigger the appropriate parser. The structure can be spidered by descending through the references, or presented to a user to browse through some kind of interface. A RESTful interface becomes a way for clients to ask for information based on what they know. They start from a known or discovered point and browse the information like you browse the Web.

This is what HATEOS refers to. The application state is transferred and discovered within the hypertext responses. Just like the browser needs to know about HTML, images, sound files, etc., a RESTful client will need to know how to parse the results of resolving a resource reference. However, the entire process is simple, constrained, scalable and flexible -- exactly the properties we want from a networked software system.

Many people build "RESTful" systems that require the clients to know beforehand what each level in a URL means. Should the information get reorganized on the server side, clients of those systems will break. Clients that truly embody HATEOS are more loosely-coupled from the servers they communicate with.

Looking Forward

We struggle daily to solve the problems of rapidly changing domains, technologies, customer demands and actionable knowledge. We spend too much time writing software to link what we learn to what we know. Objects and databases have not kept pace with the changes we experience. We need a new way of looking at the information we produce and consume that is extensible and less fragile than the solutions of the past. We need technology to help us form consensus. We should not have to achieve consensus in the form of common models before we can use our technologies.

In this article, we have introduced the series and have begun to look at how REST and Web technologies can serve as the basis of a new information-oriented architecture. We have established a naming scheme that allows us to unify references to all manner of content, services and documents. Clients can leverage the freedom to negotiate information into the form they want. As they resolve references, they can discover new content connected through new relationships.

This architectural style and the technologies surrounding the Semantic Web combine nicely to create powerful, scalable, flexible software systems. Their capacity to create Webs of Data will have as much impact on our lives as the Web has already had. This will be an information systems revolution that will turn much of what we know on its head. It will not only reduce the cost of data integration, but it will enable new business capabilities we can only begin to imagine.

We are moving into a world where information can be connected and used regardless of whether it is contained in documents, databases or is returned as the results of a RESTful service. We will be able to discover content and connect it to what we already know. We will be able to surface the data currently hidden behind databases, spreadsheets, reports and other silos. Not only will we gain access to this information, we will be able to consume it in the ways we want to.

This is one of the main, modest goals of the Semantic Web. Achieving it, as we are now able to do, is starting to change everything.

Link: http://www.infoq.com/articles/roa-rest-of-rest

Related Contents：

- JavaOne Semantic Web Panel
- Cool URIs in a RESTful World
- A Comparative Clarification: Microformats vs. RDF
- The Semantic Web and Ontological Technologies Continue to Expand
- SPARQL Update to Complete RESTful SOA Scenario

Articles

RESTful HTTP in practice

Author: Gregor Roth

This article gives a short overview about the basics of RESTful HTTP and discusses typical issues that developers face when they design RESTful HTTP applications. It shows how to apply the REST architecture style in practice. It describes commonly used approaches to name URIs, discusses how to interact with resources through the Uniform interface, when to use PUT or POST and how to support non-CRUD operations.

REST is a style, not a standard. There is neither a REST RFC, nor a REST protocol specification nor something similar. The REST architecture style has been described in the dissertation of Roy Fielding, one of the principal authors of the HTTP and URI specification. An architecture style such as REST defines a set of high-level architectures decisions which is implemented by an application. Applications which implement a dedicated architecture style will use the same patterns and other architectural elements such as caching or distribution strategies in the same way. Roy Fielding described REST as an architecture style which attempts “to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations"

Even though REST is heavily influenced by the Web-Technology, in theory the REST architecture style is not bound to HTTP. However, HTTP is the only relevant instance of the REST. For this reason this article describes REST implemented by using HTTP. Often this is called RESTful HTTP.

The idea behind RESTful HTTP is to use the existing features and capabilities of the WEB. REST does not invent new technologies, components or services. RESTful HTTP defines the principles and constrains to use the existing WEB-Standards in a better way.

Resources

Resources are the key abstractions in REST. They are the remote accessible objects of the application. A resource is a unit of identification. Everything that might be accessed or be manipulated remotely could be a resource. Resources can be static, which means the state of the resource will not change over the time. On the other side other resources can have a high degree of variance in their state over time. Both types of resources are vaild types.

For instance, the classes, shown in Figure 1, could easily be mapped to such resources. Mapping

entity classes such as Hotel or Room to resources will not be very comprehensible for object oriented designers. The same is true for mapping control classes which represent coordination, transactions, or control of other classes.

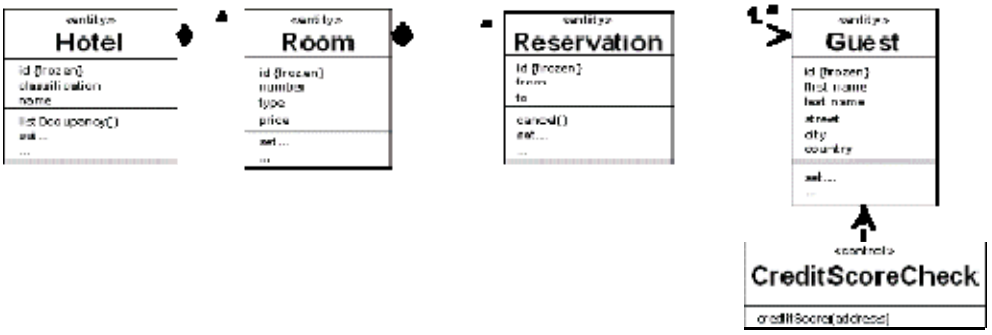


Figure 1: Example analysis model

The analysis model is a good starting point for identifying resources. However, there is not necessarily a one-to-one mapping. For instance, the <Hotel>.listOccupancy() operation can also be modelled as resources. Further more there could also be resources which represents (parts of) some entities. The primary drivers of the resource design are networking aspects and not the object model.

Any important resource is reachable through a unique identifier. RESTful HTTP uses URIs to identify resources. URIs are providing identification that is common across the Web. They contain everything the client needs to interact with the referred resource.

How to name Resource Identifiers?

Even though RESTful HTTP does not specify how a URI path have to be structured, in practice often specific naming schemas for the URI path is used. URI naming schemas help to debug and trace applications. Often a URI contains the resource type name followed by an identifier to address a dedicated resource. Such a URI will not contain verbs which indicate a business operation to process. It is only used to address resources. Figure (a1) shows an example URI of a Hotelresource. Alternatively the same Hotelcan be accessed by URI (a2). A resource can be refered by more than one URI.

```
(a1) http://localhost/hotel/656bcee2-28d2-404b-891b

(a2) http://127.0.0.1/hotel/656bcee2-28d2-404b-891b

(b) http://localhost/hotel/656bcee2-28d2-404b-891b/Room/4

(c) http://localhost/hotel/656bcee2-28d2-404b-891b/Reservation/15

(d)
http://localhost/hotel/656bcee2-28d2-404b-891b/Room/4/Reservation/15
```

```
(e)
http://localhost/hotel/656bcee2-28d2-404b-891b/Room/4/Reservation/15
v7

(f) http://localhost/hotel/656bcee2-28d2-404bv12
```

Figure 2: Examples of addressing resources

URIs can also be used by resources to establish relationships between resource representations. For instance a Hotelrepresentation will refer the assigned Room resources by using a URI, not by using a plain Room id. Using a plain id would force the caller to construct a URI by accessing the resource. The caller would not be able to access the resource without additional context knowledge such as the host name or the base URI path.

Hyperlinks are used by clients to navigate through the resources. RESTful APIs are hypertext-driven, which means by getting a Hotelrepresentation the client will be able to navigate to the assigned Room representations and the assigned Reservation representations.

In practice, classes such as shown in figure 1 will often be mapped in the sense of business objects. This means the URI stays persistent throughout the lifecycle of the business object. If an new resource is created, a new URI will be allocated. After deleting the resource the URI becomes invalid. The URI (a), (b), (c) and (d) are examples of such identifiers. On the other side a URI can also be used to referring object snapshots. For instance the URI (e) and (f) would refer such a snapshot by including a version identifier within the URI.

URIs can also addresses "sub" resources as shown in example (b), (c), (d) and (e). Often aggregated objects will be mapped to sub-resources such as the Room which is aggregated by the Hotel. Aggregated objects do not have their own lifecycle and if the parent object is deleted, all aggregated objects will also be deleted.

However, if a “sub" resource can be moved from one parent resource to another one it should not include the parent resource identifier within the URI. For instance the Reservation, shown in Figure 1 can be assigned to another Room. A Reservation resource URI which contains the Room identifier such as shown in (d) will become invalid, if the Room instance identifier changes. If such a Reservation URI is referred by another resource, this will be a problem. To avoid invalid URIs the Reservation could be addressed such as shown in (c).

Normally the resource URIs are controlled by the server. The clients do not have to understand the resource URI namespace structure to access the resource. For instance using the URI structure (c) or the URI structure (d) will have the same effects for the client.

Uniform Resource interface

To simplify the overall system architecture the REST architecture style includes the concept of a Uniform Interface. The Uniform Interface consists of a constrained set of well-defined operations to access and manipulate resources. The same interface is used regardless of the resource. If the client

interacts with a Hotelresource, a Room resource or a CreditScore resource the interface will be the same. The Uniform Interface is independent to the resource URI. No IDL-like files are required describing the available methods.

The interface of RESTful HTTP is widely used and very popular. It consists of the standard HTTP methods such as GET, PUT or POST which is used by internet browsers to retrieve pages and to send data. Unfortunately a lot of developers believe implementing a RESTful application just means to use HTTP in a direct way, which it is not. For instance the HTTP methods have to be implemented according to the HTTP specification. Using a GET method to create or to modify objects violates the HTTP specification.

Uniform Interface applied

Fielding's dissertation does not include a table, a list or something else which describes in detail when and how to use the different HTTP verbs. For the most methods such as GET or DELETE it becomes clear by reading the HTTP specification. This is not true for POST and partial updates. In practice different approaches exists to perform partial updates on resources which will be discussed below.

Table 1 list the typical usage of the most important methods GET, DELETE, PUT and POST

Important Methods	Typical Usage	Typical Status Codes	Safe?	Idempotent
GET	- retrieve a representation - retrieve a representation if modified (caching)	200 (OK) - the representation is sent in the response 204 (no content) - the resource has an empty representation 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 304 (not modified) - the resource has not been modified (caching) 400 (bad request) - indicates a bad request (e.g. wrong parameter) 404 (not found) - the resource does not exits 406 (not acceptable) - the server does not support the required representation 500 (internal server error) - generic error response	yes	yes

Important Methods	Typical Usage	Typical Status Codes	Safe?	Idempotent
		503 (Service Unavailable) - The server is currently unable to handle the request		
DELETE	- delete the resource	200 (OK) - the resource has been deleted 301 (Moved Permanently) – the resource URI has been updated 303 (See Other) - e.g. load balancing 400 (bad request) - indicates a bad request 404 (not found) - the resource does not exits 409 (conflict) - general conflict 500 (internal server error) – generic error response 503 (Service Unavailable) - The server is currently unable to handle the request	no	yes
PUT	- create a resource with client-side managed instance id - update a resource by replacing - update a resource by replacing if not modified (optimistic locking)	200 (OK) - if an existing resource has been updated 201 (created) - if a new resource is created 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 400 (bad request) - indicates a bad request 404 (not found) - the resource does not exits 406 (not acceptable) - the server does not support the required representation 409 (conflict) - general conflict 412 (Precondition Failed) e.g. conflict by performing conditional update 415 (unsupported media type) - received representation is not supported 500 (internal server error) – generic error response 503 (Service Unavailable) - Theserver is currently unable to handle the request	no	yes
	- create a resource with server-side managed	200 (OK) - if an existing resource has been updated 201 (created) - if a new resource is created		

Important Methods	Typical Usage	Typical Status Codes	Safe?	Idempotent
POST	(auto generated) instance id - create a sub-resource - partial update of a resource - partial update a resource if not modified (optimistic locking)	202 (accepted) - accepted for processing but not been completed (Async processing) 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 400 (bad request) - indicates a bad request 404 (not found) - the resource does not exits 406 (not acceptable) - the server does not support the required representation 409 (conflict) - general conflict 412 (Precondition Failed) e.g. conflict by performing conditional update 415 (unsupported media type) - received representation is not supported 500 (internal server error) - generic error response 503 (Service Unavailable) - The server is currently unable to handle the request	no	no

Table 1: Example of a Uniform Interface

Representations

Resources will always be manipulated through representations. A resource will never be transmitted over the network. Instead representations of a resource are transmitted. A representation consists of data and metadata describing the data. For instance the Content-Type header of a HTTP message is such a metadata attribute.

Figure 3 shows how to retrieve a representation by using Java. This example uses the HttpClient of the Java HTTP library xLightweb which is maintained by the author.

```
HttpClient httpClient = new HttpClient();


IHttpRequest request = new GetRequest(centralHotelURI);
IHttpResponse response = httpClient.call(request);
```

Figure 3: Java example to retrieve a representation

By performing the HTTP client's call method, an http request will be sent, which requests a

representation of the Hotel resource. The returned representation, shown in Figure 4, also includes a Content-Type header which indicates the media type of the entity-body.

```
REQUEST:

GET /hotel/656bcee2-28d2-404b-891b HTTP/1.1

Host: localhost

User-Agent: xLightweb/2.6


RESPONSE:

HTTP/1.1 200 OK

Server: xLightweb/2.6

Content-Length: 277

Content-Type: application/x-www-form-urlencoded


classification=Comfort&name=Central&RoomURI=http%3A%2F%2Flocalhost%2
Fhotel%2F

656bcee2-28d2-404b-891b%2FRoom%2F2&RoomURI=http%3A%2F%2Flocalhost%2F
hotel%2F6

56bcee2-28d2-404b-891b%2FRoom%2F1
```

Figure 4: RESTful HTTP interaction

How to support specific representations?

Sometimes only a reduced set of attributes should be received to avoid transferring large data sets. In practice, one approach to determine the attributes of a representation is to support addressing specific attributes as shown in figure 5.

```
REQUEST:

GET /hotel/656bcee2-28d2-404b-891b/classification HTTP/1.1

Host: localhost

User-Agent: xLightweb/2.6

Accept: application/x-www-form-urlencoded


RESPONSE:
```

```
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 26
Content-Type: application/x-www-form-urlencoded; charset=utf-8

classification=Comfort
```

Figure 5: Attribute filtering

The GET call, shown in figure 5, requests only one attribute. To request more than one attribute the required attributes could be separated by using a comma as shown in figure 6.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/classification,name HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded

RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 43
Content-Type: application/x-www-form-urlencoded; charset=utf-8

classification=Comfort&name=Central
```

Figure 6: Multiattribute filtering

Another way to determine the required attributes is to use a query parameter which lists the required attributes as shown in figure 7. Query parameter will also be used to define query conditions or more complex filter or query criteria.

```
REQUEST:
GET
/hotel/656bcee2-28d2-404b-891b?reqAttr=classification&reqAttr=name
HTTP/1.1
Host: localhost
```

```
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded

RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 43
Content-Type: application/x-www-form-urlencoded; charset=utf-8

classification=Comfort&name=Central
```

Figure 7: Query-String

In the examples above the server always returns a representation which is encoded by the media type application/x-www-form-urlencoded. Essentially this media type encodes an entity as a list of key-value-pairs. The key-value approach is very easy to understand. Unfortunately it will not fit well, if more complex data structures have to be encoded. Further more this media type does not support a binding of scalar data types such as Integer, Boolean or Date. For this reason often XML, JSON or Atom is used to represent resources (JSON also does not define the binding of the Date type).

```
HttpClient httpClient = new HttpClient();

IHttpRequest request = new GetRequest(centralHotelURI);
request.setHeader("Accept", "application/json");

IHttpResponse response = httpClient.call(request);

String jsonString = response.getBlockingBody().readString();
JSONObject jsonObject = (JSONObject)
JSONSerializer.toJSON(jsonString);
HotelHotel= (Hotel) JSONObject.toBean(jsonObject, Hotel.class);
```

Figure 8: Requesting a JSON representation

By setting the request accept header, the client is able to request for a specific representation encoding. Figure 8 shows how to request a representation of the media type application/json. The returned response message shown in figure 9 will be mapped to a Hotelbean by using the library

JSONlib.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/json

RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 263
Content-Type: application/json; charset=utf-8

{"classification":"Comfort",
"name":"Central",
"RoomURI":["http://localhost/hotel/656bcee2-28d2-404b-891b/Room/1",
      "http://localhost/hotel/656bcee2-28d2-404b-891b/Room/2"]}
```

Figure 9: JSON representation

How to signal errors?

What happens if the server does not support the required representation? Figure 10 shows a HTTP interaction which requests for a XML representation of the resource. If the server does not support the required representation, it will return a HTTP 406 response indicating to refuse to service the request.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: text/xml

RESPONSE:
```

```
HTTP/1.1 406 No match for accept header
Server: xLightweb/2.6
Content-Length: 1468
Content-Type: text/html; charset=iso-8859-1

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1"/>
    <title>Error 406 No match for accept header</title>
  </head>
  <body>
     <h2>HTTP ERROR: 406</h2><pre>No match for accept header</pre>
       ...
  </body>
</html>
```

Figure 10: Unsupported representation

A RESTful HTTP server application has to return the status code according to the HTTP specification. The first digit of the status code identifies the type of the result. 1xx indicates a provisional response, 2xx a successful response, 3xx a redirect, 4xx a client error and 5xx a server error. Misusing the response code or always returning a 200 response, which contains an application specific response in the body is a bad idea.

Client agents and intermediaries also evaluate the response code. For instance xLightweb's HttpClient pools persistent HTTP connections by default. After an HTTP interaction a persistent HTTP connection will be returned into an internal pool for reuse. This will only be done for healthy connection. For instance connections will not be returned if a 5xx status code is received.

Sometimes specific clients require a more precise status code. One approach to do this is to add an X-Header, which details the HTTP status code as shown in figure 11.

```
REQUEST:
POST /Guest/ HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
```

```
Content-Length: 94
Content-Type: application/x-www-form-urlencoded


zip=30314&lastName=Gump&street=42+Plantation+Street&firstName=Forest
&country=US&
city=Baytown&state=LA


RESPONSE:
HTTP/1.1 400 Bad Request
Server: xLightweb/2.6
Content-Length: 55
Content-Type: text/plain; charset=utf-8
X-Enhanced-Status: BAD_ADDR_ZIP


AddressException: bad zip code 99566
```

Figure 11: Enhanced staus code

Often the detailed error code is only necessary to diagnose programming errors. Although a HTTP status code is often less expressive than a detailed error code, in most cases they are sufficient for the client to handle the error correctly. Another approach is to include the detailed error code into the response body

PUTting or POSTing?

In contrast to popular RPC approaches the HTTP methods do not only vary in the method name. Properties such as idempotency or safety play an important role for HTTP methods. Idempotency and safety varies for the different HTTP methods.

```
HttpClient httpClient = new HttpClient();


String[] params = new String[] { "firstName=Forest",
            "lastName=Gump",
            "street=42 Plantation Street",
            "zip=30314",
```

```
            "city=Baytown",
            "state=LA",
            "country=US"};
IHttpRequest request = new PutRequest(gumpURI, params);
IHttpResponse response = httpClient.call(request);
```

Figure 12: Performing a PUT method

For instance figure 12 and 13 show a PUT interaction to create a new Guest resource. A PUT method stores the enclosed resource under the supplied Request-URI. The URI will be determined on the client-side. If the Request-URI refers to an already existing resource, this resource will be replaced by the new one. For this reason the PUT method will be used to create a new resource as well as to update an existing resource. However, by using PUT, the complete state of the resource has to be transferred. The update request to set the zip field has to include all other fields of the Guest resource such as firstName or city.

```
REQUEST:
PUT Hotel/guest/bc45-9aa3-3f22d HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 94
Content-Type: application/x-www-form-urlencoded


zip=30314&lastName=Gump&street=42+Plantation+Street&firstName=Forest
&country=US&
city=Baytown&state=LA


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 36
Content-Type: text/plain; charset=utf-8
Location: http://localhost/guest/bc45-9aa3-3f22d
```

The guest resource has been updated.

Figure 13: HTTP PUT interaction

The PUT method is idempotent. An idempotent method means that the result of a successful performed request is independent of the number of times it is executed. For instance you can execute a PUT method to update the Hotelresource as many times as you like, the result of a successful execution will always be the same. If two PUT methods occur simultaneously, one of them will win and determine the final state of the resource. The DELETE method is also idempotent. If a PUT method occurs concurrently to a DELETE method, the resourced will be updated or deleted, but nothing in between.

If you are not sure if the execution of a PUT or DELETE was successful and you did not get a status code such as 409 (Conflict) or 417 (Expectation Failed), re-execute it. No additional reliability protocols are necessary to avoid duplicated request. In general a duplicated request does not matter.

This is not true for the POST method, because the POST method is not idempotent. Take care by executing the same POST method twice. The missing idempotency is the reason why a browser always pops up a warning dialog when you retry a POST request. The POST method will be used to create a resource without determining an instance-specific id on the client-side. For instance figure 14 shows a HTTP interaction to create a Hotelresource by performing a POST method. Typically the client sends the POST request by using a URI which contains the URI base path and the resource type name.

```
REQUEST:
POST /HotelHTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 35
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Accept: text/plain

classification=Comfort&name=Central

RESPONSE:
HTTP/1.1 201 Created
Server: xLightweb/2.6
Content-Length: 40
```

```
Content-Type: text/plain; charset=utf-8
Location: http://localhost/hotel/656bcee2-28d2-404b-891b
```

the Hotelresource has been created

Figure 14: HTTP POST interaction (create)

Often the POST method will also be used to update parts of the resource. For instance sending a PUT requests which contains only the classification to update the Hotelresource violates HTTP. This is not true for the POST method. The POST method is neither idempotent nor safe. Figure 15 shows such a partial update by using a POST method.

```
REQUEST:
POST /hotel/0ae526f0-9c3d HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 19
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Accept: text/plain

classification=First+Class

RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 52
Content-Type: text/plain; charset=utf-8
```

the Hotelresource has been updated (classification)

Figure 15: HTTP POST interaction (update)

Partial update can also be performed by using the PATCH method. The PATCH method is a specialized method to apply partial modifications to a resource. A PATCH request includes a patch document which will be applied to the resource identified by the Request-URI. However, the PATCH RFC is in draft.

Using HTTP caching

To improve the scalability and to reduce the server load RESTful HTTP applications can make use of the WEB-Infrastructure caching features. HTTP recognizes caching as an integral part of the WEB infrastructure. For instance the HTTP protocol defines specific message headers to support caching. If the server sets such headers, clients such as HTTP clients or Web caching proxies will be able to support efficient caching strategies.

```
HttpClient httpClient = new HttpClient();
httpClient.setCacheMaxSizeKB(500000);

IHttpRequest request = new GetRequest(centralHotelURI +
"/classification");
request.setHeader("Accept", "text/plain");

IHttpResponse response = httpClient.call(request);
String classification = response.getBlockingBody.readString();

// ... sometime later re-execute the request
response = httpClient.call(request);
classification = response.getBlockingBody.readString();
```

Figure 16: Client-side caching interaction

For instance figure 16 shows a repeated GET call. By setting the cache max size larger than 0 the caching support of the HttpClient is activated. If the response contains freshness headers such as Expires or Cache-Control: max-age, the response will be cached by the HttpClient. These headers tell how long the associated representation is fresh for. If the same request is performed within this period of time, the HttpClient will serve the request using the cache and avoid a repeated network call. On the network, shown in figure 17, only one HTTP interaction in total occurs. Caching intermediaries such as WEB proxies implement the same behaviour. In this case the cache can be shared between different clients.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/classification HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
```

```
Accept: text/plain

RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Cache-Control: public, max-age=60
Content-Length: 26
Content-Type: text/plain; charset=utf-8
```

comfort

Figure 17: HTTP response including an expire header

The expiration model works very well for static resources. Unfortunately, this is not true for dynamic resources where changes in resource state occur frequently and unpredictably. HTTP supports caching dynamic resources by validation headers such as Last-Modified and ETag. In contrast to the expiration model, the validation model do not save a network request. However, executing a conditional GET can safe expensive operations to generate and transmit a response body. The conditional GET shown in figure 18 (2. request) contains an additional Last-Modified header which holds the last modified date of the cached response. If the resource is not changed, the server will reply with a 304 (Not Modified) response.

```
1. REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/Reservation/1 HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded

1. RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 252
Content-Type: application/x-www-form-urlencoded
Last-Modified: Mon, 01 Jun 2009 08:56:18 GMT
```

```
from=2009-06-01T09%3A49%3A09.718&to=2009-06-05T09%3A49%3A09.718&gues
tURI=
http%3A%2F%2Flocalhost%2Fguest%2Fbc45-9aa3-3f22d&RoomURI=http%3A%2F%
2F
localhost%2Fhotel%2F656bcee2-28d2-404b-891b%2FRoom%2F1


2. REQUEST:
GET /hotel/0ae526f0-9c3d/Reservation/1 HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded
If-Modified-Since: Mon, 01 Jun 2009 08:56:18 GMT


2. RESPONSE:
HTTP/1.1 304 Not Modified
Server: xLightweb/2.6
Last-Modified: Mon, 01 Jun 2009 08:56:18 GMT
```

Figure 18: Validation-based caching

Do not store application state on the server-side

A RESTful HTTP interaction has to be stateless. This means each request contains all information which is required to process the request. The client is responsible for the application state. A RESTful server does not have to retain the application state between requests. The Server is responsible for the resource state not for the application state. Servers and intermediaries are able to understand the request and response in isolation. Web caching proxies do have all the information to handle the messages correctly and to manage their caches.

This stateless approach is a fundamental principle to implement high-scalable, high-available applications. In general statelessness enables that each client request can be served by different servers. A server can be replaced by another one for each request. As traffic increases, new servers are added. If a server fails, it will be remove from the cluster. For a more detailed explanation on load balancing and fail-over refer to the article Server load balancing architectures.

Supporting non-CRUD operations

Often developers wonder how to map non-CRUD (Create-Read-Update-Delete) operations to resources. It is obviously that Create, Read, Update and Delete operations will map very well to resource methods. However, RESTful HTTP is not limited to CRUD-oriented applications.

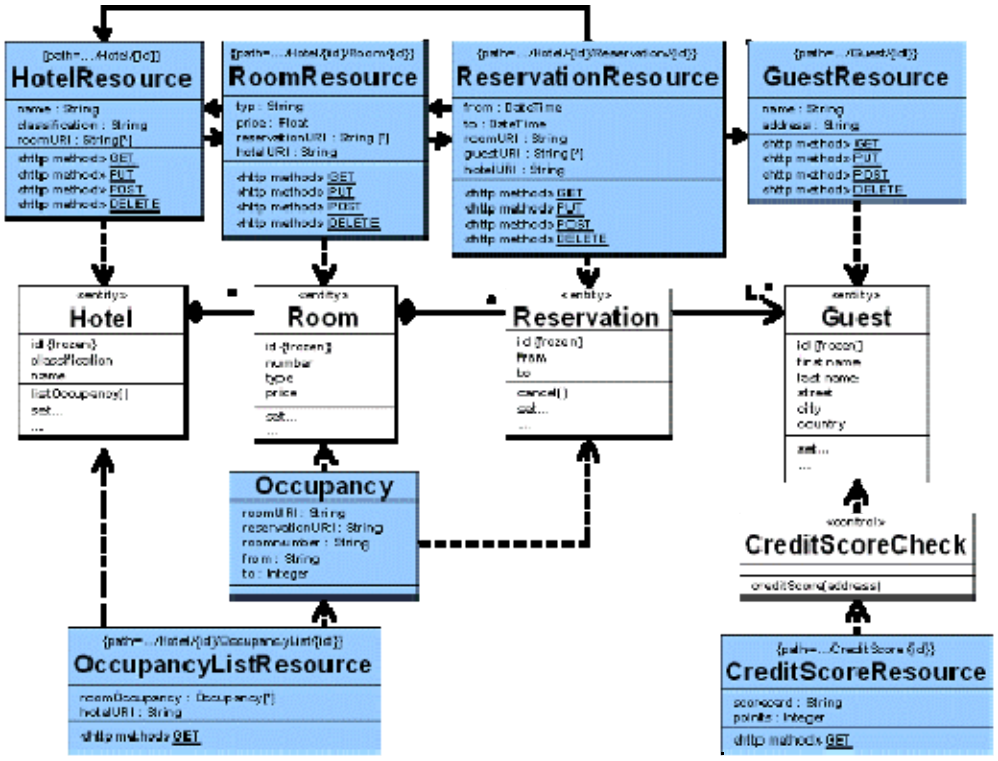


Figure 19: RESTful HTTP Resources

For instance the creditScoreCheck class shown in figure 19 provides a non-CRUD operation creditScore(...) which consumes an address, calculates the score and returns it. Such an operation can be implemented by a CreditScoreResource which represents the result of the computation. Figure 20 shows the GET call which passes over the address to process and retrieves the CreditScoreResource representation. The query parameters are used to identify the CreditScoreResource. The GET method is safe and cacheable which fits very well to non-functional behaviour of the CreditScore Check's creditScore(...) method. The result of the score calculation can be cached for a period of time. As shown in figure 20 the response includes a cache header to enable clients and intermediaries to cache the response.

```
REQUEST:

GET
/CreditScore/?zip=30314&lastName=Gump&street=42+Plantation+Street&
```

```
        firstName=Forest&country=US&city=Baytown&state=LA HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded

RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
Cache-Control: public, no-transform, max-age=300

scorecard=Excellent&points=92
```

Figure 20: Non-CRUD HTTP GET interaction

This example also shows the limit of the GET method. Although the HTTP specification does not specify any maximum length of a URL, practical limits are imposed by clients, intermediaries and servers. For this reason sending a large entity by using a GET query-parameter can fail caused by intermediary and servers which limits the URL length.

An alternative solution is performing a POST method which will also be cacheable, if indicated. As shown in figure 21 first a POST request will be performed to create a virtual resource CreditScoreResource. The input address data is encoded by the mime type text/card. After calculating the score the server sends a 201 (created) response which includes the URI of the created CreditScoreResource. The POST response is cacheable if indicated as shown in the example. By performing a GET request the credit score will be fetched. The GET response also includes a cache control header. If the client re-executes these two requests immediately, all responses can be served by the cache.

```
1. REQUEST:
POST /CreditScore/ HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 198
Content-Type: text/x-vcard
```

```
Accept: application/x-www-form-urlencoded


BEGIN:VCARD
VERSION:2.1
N:Gump;Forest;;;;
FN:Forest Gump
ADR;HOME:;;42 Plantation St.;Baytown;LA;30314;US
LABEL;HOME;ENCODING=QUOTED-PRINTABLE:42 Plantation St.=0D=0A30314
Baytown=0D=0ALA US
END:VCARD


1. RESPONSE:
HTTP/1.1 201 Created
Server: xLightweb/2.6
Cache-Control: public, no-transform, max-age=300
Content-Length: 40
Content-Type: text/plain; charset=utf-8
Location: http://localhost/CreditScore/l00000001-l0000005c

the credit score resource has been created


2. REQUEST:
GET /CreditScore/l00000001-l0000005c HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6

2. RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 31
```

```
Content-Type: application/x-www-form-urlencoded
Cache-Control: public, no-transform, max-age=300

scorecard=Excellent&points=92
```

Figure 21: Non-CRUD HTTP POST interaction

There are also some variants of this approach. Instead of returning a 201 response a 301 (Moved Permanently) redirect response could be returned. The 301 redirect response is cacheable by default. Another variant which avoids a second request is to add the representation of the newly create CreditScoreResource to the 201 response.

Conclusion

Most SOA architectures such as SOAP or CORBA try to map the class model, such as shown in Figure 1, more or less one-to-one for remote access. Typically, such SOA architectures are highly focused on transparent mapping of programming language objects. The mapping is easy to understand and very traceable. However aspects such as distribution and scalability are reduced to playing a second role.

In contrast, the major driver of the REST architecture style is distribution and scalability. The design of a RESTful HTTP interface is driven by networking aspects, not by language binding aspects. RESTful HTTP does not try to encapsulate aspects, which are difficult to hide such as network latency, network robustness or network bandwidth.

RESTful HTTP applications use the HTTP protocol in a direct way without any abstraction layer. There are no REST specific data field such as error fields or security token fields. RESTful HTTP applications will just use the capability of the WEB. Designing RESTful HTTP interfaces means that the remote interface designer has to think in HTTP. Often this leads to an additional step within the development cycle.

However, RESTful HTTP allows implementing very scalable and robust applications. Especially companies which provide web applications for a very large user group such as WebMailing or SocialNetworking applications can benefit from the REST architecture style. Often such applications have to scale very high and very fast. Further more, such companies often have to run their application on a low-budget infrastructure which is built on widely-used standard components and software.

About the author

Gregor Roth, creator of the xLightweb HTTP library, works as a software architect at United Internet group, a leading European Internet Service Provider to which GMX, 1&1, and Web.de belong. His areas of interest include software and system architecture, enterprise architecture management,

object-oriented design, distributed computing, and development methodologies.

Literature

Roy Fielding - Architectural Styles and the Design of Network-based Software Architectures

Steve Vinoski - REST Eye for the SOA Guy

Steve Vinoski - Presentation: Steve Vinoski on REST, Reuse and Serendipity

Stefan Tilkov - A Brief Introduction to REST

Wikipedia - Fallacies of Distributed Computing

Gregor Roth - Server load balancing architectures

Gregor Roth - Asynchronous HTTP and Comet architectures

JSON-lib

xLightweb

Link: http://www.infoq.com/articles/designing-restful-http-apps-roth

Related Contents：

- 80legs Is a Web Crawling Service
- Practical Advice for SOA Implementers
- RPC and its Offspring: Convenient, Yet Fundamentally Flawed
- HTTP Status Report
- Building Scalable Web Services

Articles

How to GET a Cup of Coffee

Author: Jim Webber, Savas Parastatidis & Ian Robinson

We are used to building distributed systems on top of large middleware platforms like those implementing CORBA, the Web Services protocols stack, J2EE, etc. In this article, we take a different approach, treating the protocols and document formats that make the Web tick as an application platform, which can be accessed through lightweight middleware. We showcase the role of the Web in application integration scenarios through a simple customer-service interaction scenario. In this article, we use the Web as our primary design philosophy to distil and share some of the thinking in our forthcoming book “GET /connected - Web-based integration” (working title).

Introduction

The integration domain as we know it is changing. The influence of the Web and the trend towards more agile practices are challenging our notions of what constitutes good integration. Instead of being a specialist activity conducted in the void between systems – or even worse, an afterthought – integration is now an everyday part of successful solutions.

Yet, the impact of the Web is still widely misunderstood and underestimated in enterprise computing. Even those who are Web-savvy often struggle to understand that the Web isn't about middleware solutions supporting XML over HTTP, nor is it a crude RPC mechanism. This is a shame because the Web has much more value than simple point-to-point connectivity; it is in fact a robust integration platform.

In this article we'll showcase some interesting uses of the Web, treating it as a pliant and robust platform for doing very cool things with enterprise systems. And there is nothing that typifies enterprise software more than workflows…

Why Workflows?

Workflows are a staple of enterprise computing, and have been implemented in middleware practically forever (at least in computing terms). A workflow structures work into a number of discrete steps and the events that prompt transitions between steps. The overarching business process implemented by a workflow often spans several enterprise information systems, making workflows fertile ground for integration work.

Starbucks: Standard generic coffee deserves standard generic integration

If the Web is to be a viable technology for enterprise (and wider) integration, it has to be able to support workflows – to reliably coordinate the interactions between disparate systems to implement some larger business capability.

To do justice to a real-world workflow, we'd no doubt have to address a wealth of technical and domain-specific details, which would likely obscure the aim of this article, so we've chosen a more accessible domain to illustrate how Web-based integration works: Gregor Hohpe's Starbucks coffee shop workflow. In his popular blog posting, Gregor describes how Starbucks functions as a decoupled revenue-generating pipeline:

> "Starbucks, like most other businesses is primarily interested in maximizing throughput of orders. More orders equals more revenue. As a result they use asynchronous processing. When you place your order the cashier marks a coffee cup with your order and places it into the queue. The queue is quite literally a queue of coffee cups lined up on top of the espresso machine. This queue decouples cashier and barista and allows the cashier to keep taking orders even if the barista is backed up for a moment. It allows them to deploy multiple baristas in a Competing Consumer scenario if the store gets busy.

While Gregor prefers EAI techniques like message-oriented middleware to model Starbucks, we'll model the same scenario using Web resources – addressable entities that support a uniform interface. In fact, we'll show how Web techniques can be used with all the dependability associated with traditional EAI tools, and how the Web is much more than XML messaging over a request/response protocol!

We'll apologise in advance for taking liberties with the way Starbucks works because our goal here isn't to model Starbucks completely accurately, but to illustrate workflows with Web-based services. So with belief duly suspended, let's jump in.

Stating the Obvious

Since we're talking about workflows, it makes sense to understand the states from which our workflows are composed, together with the events that transition the workflows from state to state. In our example, there are two workflows, which we've modelled as state machines. These workflows run concurrently. One models the interaction between the customer and the Starbucks service as shown in Figure 1 the other captures the set of actions performed by a barista as per Figure 2.

In the customer workflow, customers advance towards the goal of drinking some coffee by interacting with the Starbucks service. As part of the workflow, we assume that the customer places an order, pays, and then waits for their drink. Between placing and paying for the order, the customer can usually amend it – by, for example, asking for semi-skimmed milk to be used.

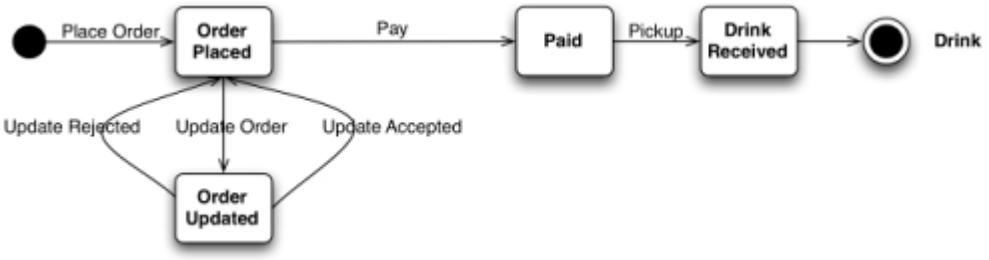


Figure1 The Customer State Machine

The barista has his or her own state machine, though it's not visible to the customer; it's private to the service's implementation. As shown in Figure 2, the barista loops around looking for the next order to be made, preparing the drink, and taking the payment. An instance of the loop can begin when an order is added to the barista's queue. The outputs of the workflow are available to the customer when the barista finishes the order and releases the drink.

Figure 2 The Barista's State Machine

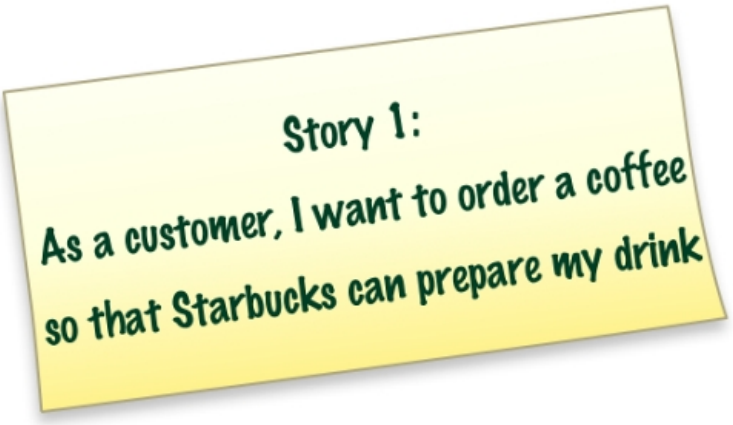
Although all of this might seem a million miles away from Web-based integration, each transition in our two state machines represents an interaction with a Web resource. Each transition is the combination of a HTTP verb on a resource via its URI causing state changes.

> GET and HEAD are special cases since they don't cause state transitions. Instead they allow us to inspect the current state of a resource.

But we're getting ahead of ourselves. Thinking about state machines and the Web isn't easy to swallow in one big lump. So let's revisit the entire scenario from the beginning, look at it in a Web context, and proceed one step at a time.

The Customer's Viewpoint

We'll begin at the beginning, with a simple story card that kick-starts the whole process:



This story contains a number of useful actors and entities. Firstly, there's the customer actor, who is the obvious consumer of the (implicit) Starbucks service. Secondly, there are two interesting entities (coffee and order), and an interesting interaction (ordering), which starts our workflow.

To submit an order to Starbucks, we simply POST a representation of an order to the well-known Starbucks ordering URI, which for our purposes will be http://starbucks.example.org/order.

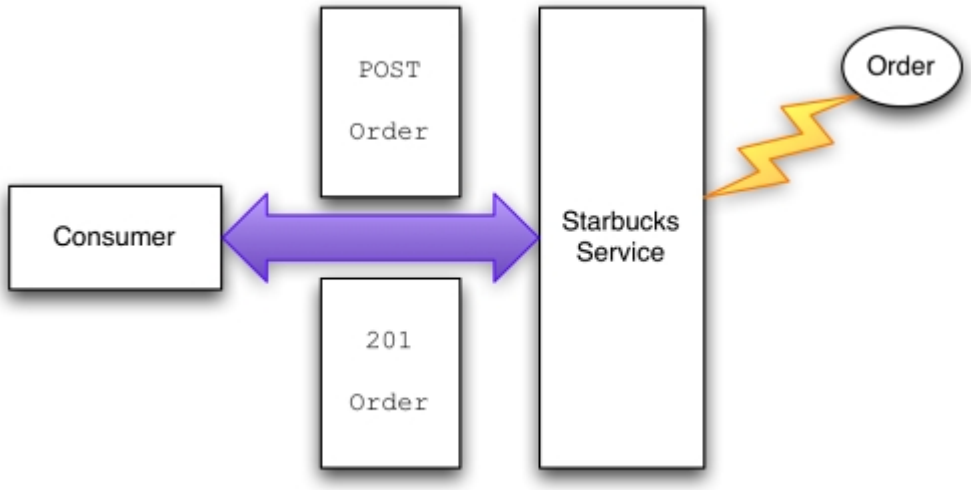


Figure 3 Ordering a coffee

Figure 3 shows the interaction to place an order with Starbucks. Starbucks uses an XML dialect to represent entities from its domain; interestingly, this dialect also allows information to be embedded so that customers can progress through the ordering process – as we'll see shortly. On the wire the act of posting looks something like Figure 4.

> "In the human Web, consumers and services use HTML as a representation format. HTML has its own particular semantics, which are understood and adopted by all browsers: <a/>, for example, means "an anchor that links to another document or to a bookmark within the same document." The consumer application – the Web browser – simply renders the HTML, and the state machine (that's you!) follows links using GET and POST. In Web-based

integration the same occurs, except the services and their consumers not only have to agree on the interaction protocols, but also on the format and semantics of the representations.

```
POST /order HTTP 1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
</order>
```

Figure 4 POSTing a drinks order

The Starbucks service creates an order resource, and then responds to the consumer with the location of this new resource in the Location HTTP header. For convenience, the service also places the representation of the newly created order resource in the response. The response looks something like .

```
201 Created
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <cost>3.00</cost>
  <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

Figure 5 Order created, awaiting payment

The 201 Created status indicates that Starbucks successfully accepted the order. The Location header gives the URI of the newly created order. The representation in the response body contains confirmation of what was ordered along with the cost. In addition, this representation contains the URI of a resource with which Starbucks expects us to interact to make forward progress with the customer workflow; we'll use this URI later.

Note that the URI is contained in a <next/> tag, not an HTML <a/> tag. <next/> is here meaningful in the context of the customer workflow, the semantics of which have been agreed a priori.

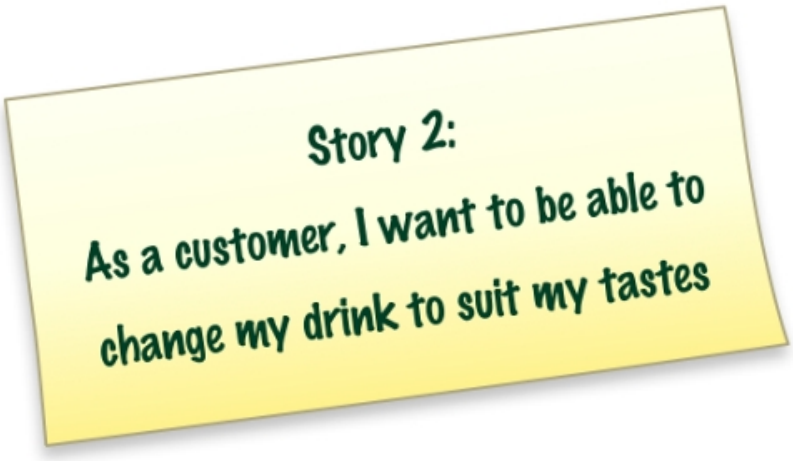
We've already seen that the 201 Created status code indicates the successful creation of a resource. We'll need a handful of other useful codes both for this example and for Web-based integration in general:

200 OK - *This is what we like to see: everything's fine; let's keep going.* 201 Created - *We've just created a resource and everything's fine.*

202 Accepted - *The service has accepted our request, and invites us to poll a URI in the Location header for the response. Great for asynchronous processing.*
303 See Other - *We need to interact with a different resource. We're probably still OK.*
400 Bad Request - *We need to reformat the request and resubmit it.*
404 Not Found - *The service is far too lazy (or secure) to give us a real reason why our request failed, but whatever the reason, we need to deal with it.*
409 Conflict - *We tried to update the state of a resource, but the service isn't happy about it. We'll need to get the current state of the resource (either by checking the response entity body, or doing a GET) and figure out where to go from there.*
412 Precondition Failed - *The request wasn't processed because an Etag, If-Match or similar guard header failed evaluation. We need to figure out how to make forward progress.*
417 Expectation Failed - *You did the right thing by checking, but please don't try to send that request for real.*
500 Internal Server Error - *The ultimate lazy response. The server's gone wrong and it's not telling why. Cross your fingers…*

Updating an Order

One of the nice things about Starbucks is you can customise your drink in a myriad of different ways. In fact, some of the more advanced customers would be better off ordering by chemical formula, given the number of upgrades they demand! But let's not be that ambitious – at least not to start with. Instead, we'll look at another story card:



Looking back on Figure 4, it's clear we made a significant error: for anyone that really likes coffee, a single shot of espresso is going to be swamped by gallons of hot milk. We're going to have to change that. Fortunately, the Web (or more precisely HTTP) provides support for such changes, and so does our service.

Firstly, we'll make sure we're still allowed to change our order. Sometimes the barista will be so fast our coffee's been made before we've had a chance to change it – and then we're stuck with a cup of hot coffee-flavoured milk. But sometimes the barista's a little slower, which gives us the opportunity

to change the order before the barista processes it. To find out if we can change the order, we ask the resource what operations it's prepared to process using the HTTP OPTIONS verb, as shown on the wire in Figure 6.

Request	Response
OPTIONS /order/1234 HTTP 1.1 Host: starbucks.example.org	200 OK Allow: GET, PUT

Figure6 Asking for OPTIONS

From Figure 6 we see that the resource is readable (it supports GET) and it's updatable (it supports PUT). As we're good citizens of the Web, we can, optionally, do a trial PUT of our new representation, testing the water using the Expect header before we do a real PUT – like in Figure 7.

Request	Response
PUT /order/1234 HTTP 1.1 Host: starbucks.example.com Expect: 100-Continue	100 Continue

Figure 7 Look before you leap!

If it had no longer been possible to change our order, the response to our “look before you leap” request in Figure 7 would have been 417 Expectation Failed. But here the response is 100 Continue, which allows us to try to PUT an update to the resource with an additional shot of espresso, as shown in Figure 8. PUTting an updated resource representation effectively changes the existing one. In this instance PUT lodges a new description with an <additions/> element containing that vital extra shot.

Although partial updates are the subject of deep philosophical debates within the REST community, we take a pragmatic approach here and assume that our request for an additional shot is processed in the context of the existing resource state. As such there is little point in moving the whole resource representation across the network for each operation and so we transmit deltas only.

```
PUT /order/1234 HTTP 1.1
Host: starbucks.example.com
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <additions>shot</additions>
</order>
```

Figure 8 Updating a resource's state

If we're successfully able to PUT an update to the new resource state, we get a 200 response from the server, as shown in Figure 9.

```
200 OK
Location: http://starbucks.example.com/order/1234
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <additions>shot</additions>
  <cost>4.00</cost>
  <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

Figure 9 Successfully updating the state of a resource

Checking OPTIONS and using the Expect header can't totally shield us from a situation where a change at the service causes subsequent requests to fail. As such we don't mandate their use, and as good Web citizens we're going to handle 405 and 409 responses anyway.

OPTIONS *and especially using the* Expect *header should be considered optional steps.*

Even with our judicious use of Expect and OPTIONS, sometimes our PUT will fail; after all, we're in a race with the barista – and sometimes those guys just fly!

If we lose the race to get our extra shot, we'll learn about it when we try to PUT the updates to the resource. The response in Figure 10 is typical of what we can expect. 409 Conflict indicates the resource is in an inconsistent state to receive the update. The response body shows the difference between the representation we tried to PUT and the resource state on the server side. In coffee terms it's too late to add the shot – the barista's already pouring the hot milk.

```
409 Conflict

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <cost>3.00</cost>
    <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

Figure 10 Losing a race

We've discussed using Expect and OPTIONS to guard against race conditions as much as possible. Besides these, we can also attach If-Unmodified-Since or If-Match headers to our PUT to convey our intentions to the receiving service. If-Unmodified-Since uses the timestamp and If-Match the ETag1 of the original order. If the order hasn't changed since we created it – that is, the barista hasn't started preparing our coffee yet – then the change will be processed. If the order has changed, we'll get a 412 Precondition Failed response. If we lose the race, we're stuck with milky coffee, but at least we've not transitioned the resource to an inconsistent state.

There are a number of patterns for consistent state updates using the Web. HTTP PUT is idempotent, which takes much of the intricate work out of updating state, but there are still choices that need to be made. Here's our recipe for getting updates right:

1. Ask the service if it's still possible to PUT *by sending* OPTIONS*. This step is optional. It gives clients a clue about which verbs the server supports for the resource at the time of asking, but there are no guarantees the service will support those same verbs indefinitely.*

2. Use an If-Unmodified-Since *or* If-Match *header to help the server guard against executing an unnecessary* PUT*. You'll get a* 412 Precondition Failed *if the* PUT *subsequently fails. This approach depends either on slowly changing resources (1 second granularity) for* If-Unmodified-Since *or support for ETags for* If-Match*.*

3. Immediately PUT *the update and deal with any* 409 Conflict *responses. Even if we use (1) and (2), we may have to deal with these responses, since our guards and checks are optimistic in nature.*

The W3C has a non-normative note *on detecting and dealing with inconsistent updates that argues for using ETag. ETags are our preferred approach.*

After all that hard work updating our coffee order, it seems only fair that we get our extra shot. So for now let's go with our happy path, and assume we managed to get our additional shot of espresso. Of course, Starbucks won't hand our coffee over unless we pay (and it turns out they've already hinted as much!), so we need another story:

Remember the <next/> element in the response to our original order? This is where Starbucks embedded information about another resource in the order representation. We saw the tag earlier, but chose to ignore it while correcting our order. But now it's time to look more closely at it:

```
<next xmlns="http://example.org/state-machine"
  rel="http://starbucks.example.org/payment"
  uri="https://starbucks.example.com/payment/order/1234"
  type="application/xml"/>
```

There are a few aspects to the next element worth pointing out. First is that it's in a different namespace because state transitions are not limited to Starbucks. In this case we've decided that such transition URIs should be held in a communal namespace to facilitate re-use (or even eventual standardisation).

Then, there's the embedded semantic information (a private microformat, if you like) in the rel attribute. Consumers that understand the semantics of the http://starbucks.example.org/payment string can use the resource identified by the uri attribute to transition to the next state (payment) in the workflow.

The uri in the <next/> element points to a payment resource. From the type attribute, we already know the expected resource representation is XML. We can work out what to do with the payment resource by asking the server which verbs that resource supports using OPTIONS.

> “Microformats are a way to embed structured, semantically-rich data inside existing documents. Microformats are most common in the human readable Web, where they are used to add structured representations of information like calendar events to Web pages. However, they can just as readily be turned to integration purposes. Microformat terminology is agreed by the microformats community, but we are at liberty to create our own private microformats for domain-specific semantic markup.

Innocuous as they seem, simple links like the one of Figure 10 are the crux of what the REST community rather verbosely calls “Hypermedia as the engine of application state.” More simply, URIs represent the transitions within a state machine. Clients operate application state machines, like the ones we saw at the beginning of this article, by following links.

Don't be surprised if that takes a little while to sink in. One of the most surprising things about this model is the way state machines and workflows gradually describe themselves as you navigate through them, rather than being described upfront through WS-BPEL or WS-CDL. But once your brain has stopped somersaulting, you'll see that following links to resources allows us to make forward progress in our application's various states. At each state transition the current resource representation includes links to the next set of possible resources and the states they represent. And because those next resources are just Web resources, we already know what to do with them.

Our next step in the customer workflow is to pay for our coffee. We know the total cost from the <cost/> element in the order, but before we send payment to Starbucks we'll ask the payment resource how we're meant to interact with it, as shown in Figure 11.

> “How much upfront knowledge of a service does a consumer need? We've already suggested that services and consumers will need to agree the semantics of the representations they exchange prior to interacting. Think of these representation formats as a set of possible states and transitions. As a consumer interacts with a service, the service chooses states and

transitions from the available set and builds the next representation. The process – the "how" of getting to a goal – is discovered on the fly; what gets wired together as part of that process is, however, agreed upfront.

Consumers typically agree the semantics of representations and transitions with a service during design and development. But there's no guarantee that as service evolves, it won't confront the client with state representations and transitions the client had never anticipated but knows how to process – that's the nature of the loosely coupled Web. Reaching agreement on resource formats and representations under these circumstances is, however, outside the scope of this article.

Our next step is to pay for our coffee. We know the total cost of our order from the <cost> element embedded in the order representation, and so our next step is to send a payment to Starbucks so the barista will hand over the drink. Firstly we'll ask the payment resource how we're meant to interact with it, as shown in Figure 11.

Request	**Response**
OPTIONS/payment/order/1234 HTTP 1.1 Host: starbucks.example.com	Allow: GET, PUT

Figure 11 Figuring out how to pay

The response indicates we can either read (via GET) the payment or update it (via PUT). Knowing the cost, we'll go ahead and PUT our payment to the resource identified by the payment link. Of course, payments are privileged information, so we'll protect access to the resource by requiring authentication2.

Request
PUT /payment/order/1234 HTTP 1.1 Host: starbucks.example.com Content-Type: application/xml Content-Length: ... Authorization: Digest username="Jane Doe" realm="starbucks.example.org" nonce="..." uri="payment/order/1234" qop=auth nc=00000001 cnonce="..." reponse="..." opaque="..." <payment xmlns="http://starbucks.example.org/">

```
  <cardNo>123456789</cardNo>
  <expires>07/07</expires>
  <name>John Citizen</name>
  <amount>4.00</amount>
</payment>
```

Response

```
201 Created
Location: https://starbucks.example.com/payment/order/1234
Content-Type: application/xml
Content-Length: ...

<payment xmlns="http://starbucks.example.org/">
  <cardNo>123456789</cardNo>
  <expires>07/07</expires>
  <name>John Citizen</name>
  <amount>4.00</amount>
</payment>
```

Figure 12 Paying the bill

For successful payments, the exchange shown in Figure 12 is all we need. Once the authenticated PUT has returned a 201 Created response, we can be happy the payment has succeeded, and can move on to pick up our drink.

But things can go wrong, and when money is at stake we'd rather things either didn't go wrong or are recoverable when they do3. A number of things can obviously go wrong with our payment:

- We can't connect to the server because it is down or unreachable;
- The connection to the server is severed at some point during the interaction;
- The server returns an error status in the 4xx or 5xx range.

Fortunately, the Web helps us in each of these scenarios. For the first two cases (assuming the connectivity issue is transient), we simply PUT the payment again until we receive a successful response. We can expect a 200 response if a prior PUT had in fact succeeded (effectively an acknowledgement of a no-op from the server) or a 201 if the new PUT eventually succeeds in lodging the payment. The same holds true in the third case where the server responds with a 500, 503 or 504 response code.

Status codes in the 4xx range are trickier, but they still indicate how to make forward progress. For example, a 400 response indicates that we PUT something the server doesn't understand, and should rectify our payload before PUTing it again. Conversely, a 403 response indicates that the server understood our request but is refusing to fulfil it and doesn't want us to re-try. In such cases

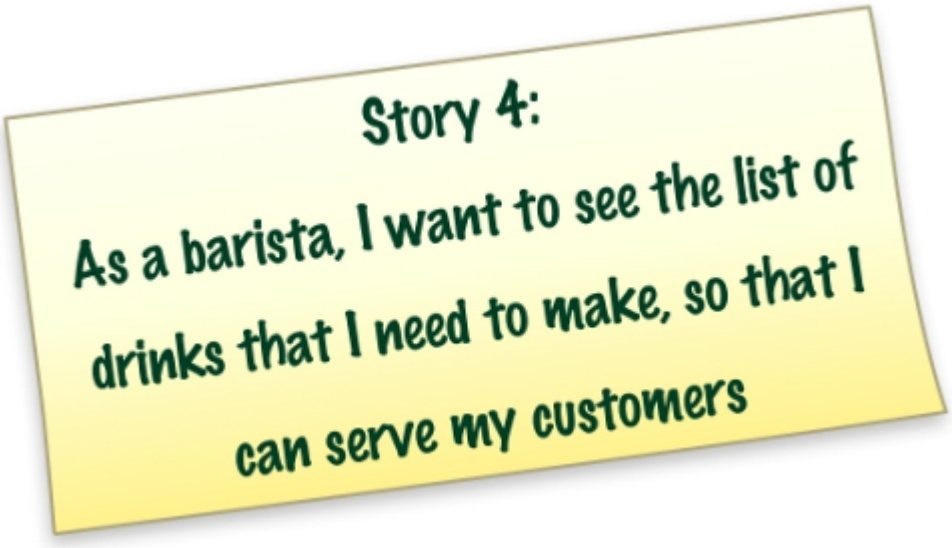
we'll have to look for other state transitions (links) in the response payload to make alternative forward progress.

> "We've used status codes several times in this example to guide the client towards its next interaction with the service. Status codes are semantically rich acknowledgments. By implementing services that produce meaningful status codes and clients that know how to handle them, we can layer a coordination protocol on top of HTTP's simple request-response mechanism, adding a high degree of robustness and reliability to distributed systems.

Once we've paid for our drink we've reached the end of our workflow, and the end of the story as far as the consumer goes. But it's not the end of the whole story. Let's now go inside the service boundary, and look at Starbucks' internal implementation.

The Barista's Viewpoint

As customers we tend to put ourselves at the centre of the coffee universe, but we're not the only consumers of a coffee service. We know already from our "race" with the barista that the service serves at least one other set of interested parties, not the least of which is the barista. In keeping with our incremental delivery style, it's time for another story card.



Lists of drinks are easily modelled using Web formats and protocols. Atom feeds are a perfectly good format for lists of practically anything, including outstanding coffee orders, so we'll adopt them here. The barista can access the Atom feed with a simple GET on the feed's URI, which for outstanding orders is http://starbucks.example.org/orders in Figure 13.

```
200 OK
Expires: Thu, 12Jun2008 17:20:33 GMT
Content-Type: application/atom+xml
Content-Length: ...

<?xml version="1.0" ?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Coffees to make</title>
  <link rel="alternate"
    uri="http://starbucks.example.org/orders"/>
  <updated>2008-06-10T19:18:43Z</updated>
  <author><name>Barista System</name></author>
  <id>urn:starkbucks:barista:coffees-to-make</id>

  <entry>
    <link rel="alternate" type="application/xml"
      uri="http://starbucks.example.org/order/1234"/>
    <id>http://starbucks.example.org/order/1234</id>
      ...
  </entry>
 ...
</feed>
```

Figure 13 Atom feed for drinks to be made

Starbucks is a busy place and the Atom feed at /orders is updated frequently, so the barista will need to poll it to stay up to date. Polling is normally thought of as offering low scalability; the Web, however, supports an extremely scalable polling mechanism – as we'll see shortly. And with the sheer volume of coffees being manufactured by Starbucks every minute, scaling to meet load is an important issue.

We have two conflicting requirements here. We want baristas to keep up-to-date by polling the order feed often, but we don't want to increase the load on the service or unnecessarily increase network traffic. To avoid crushing our service under load, we'll use a reverse proxy just outside our service to cache and serve frequently accessed resource representations, as shown in Figure 14.

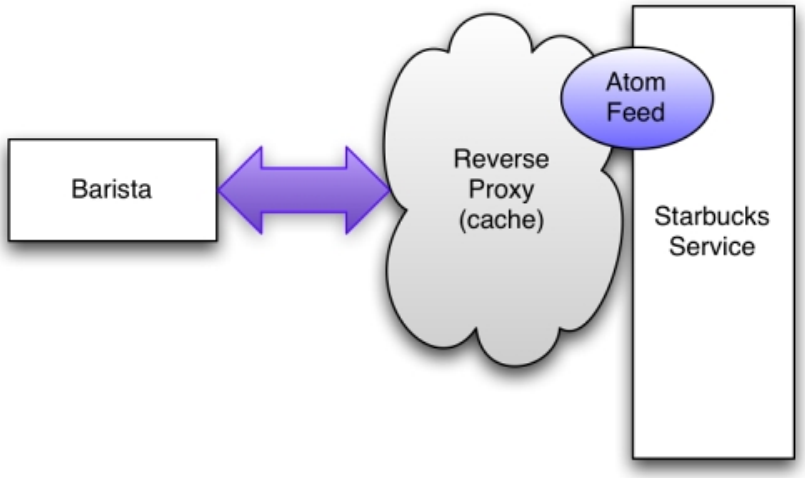


Figure 14 Caching for scalability

For most resources – especially those that are accessed widely, like our Atom feed for drinks – it makes sense to cache them outside of their host services. This reduces server load and improves scalability. Adding Web caches (reverse proxies) to our architecture, together with caching metadata, allows clients to retrieve resources without placing load on the origin server.

> A positive side effect of caching is that it masks intermittent failures of the server and helps crash recovery scenarios by improving the availability of resource state. That is, the barista can keep working even if the Starbucks service fails intermittently since the order information will have been cached by a proxy. And if the barista forgets an order (crashes) then recovery is made easier because the orders are highly available.

Of course, caching can keep old orders around longer than needed, which is hardly ideal for a high-throughput retailer like Starbucks. To make sure that cached orders are cleared, the Starbucks service uses the Expires header to declare how long a response can be cached. Any caches between the consumer and service (should) honour that directive and refuse to serve stale orders4, instead forwarding the request onto the Starbucks service, which has up-to-date order information.

The response in Figure 13 sets the Expires header on our Atom feed so that drinks turn stale 10 seconds into the future. Because of this caching behaviour, the server can expect at most 6 requests per minute, with the remainder handled by the cache infrastructure. Even for a relatively poorly performing service, 6 requests per minute is a manageable workload. In the happiest case (from Starbucks' point of view) the barista's polling requests are answered from a local cache, resulting in no increased network activity or server load.

In our example, we use only one cache to help scale-out our master coffee list. Real Web-based scenarios, however, may benefit from several layers of caching. Taking advantage of existing Web caches is critical for scalability in high volume situations.

> The Web trades latency for massive scalability. If you have a problem domain that is highly sensitive to latency (e.g. foreign exchange trading), then Web-based solutions are not a great idea. If, however, you can accept latency in the order of seconds, or even minutes or hours, then the Web is likely a suitable platform.

Now that we've addressed scalability, let's return to more functional concerns. When the barista begins to prepare our coffee, the state of the order should change so that no further updates are allowed. From the point of view of a customer, this corresponds to the moment we're no longer allowed to PUT updates of our order (as in Figure 6, Figure 7, Figure 8, Figure 9, and Figure 10).

Fortunately there is a well-defined protocol that we can use for this job: the Atom Publishing Protocol (also known as APP or AtomPub). AtomPub is a Web-centric (URI-based) protocol for managing entries in Atom feeds. Let's take a closer look at the entry representing our coffee in the /orders Atom feed.

```
<entry>
  <published>2008-06-10T19:18:43Z </published>
  <updated>2008-06-10T19:20:32Z</updated>
  <link rel="alternate" type="application/xml"
    uri="http://starbucks.example.org/order/1234"/>
  <id>http://starbucks.example.org/order/1234</id>
  <content type="text+xml">
    <order xmlns="http://starbucks.example.org/">
      <drink>latte</drink>
      <additions>shot</additions>
      <cost>4.00</cost>
    </order>
    <link rel="edit"
      type="application/atom+xml"
      href="http://starbucks.example.org/order/1234/>
  ...
  </content>
</entry>
```

Figure 15 Atom entry for our coffee order

The XML in Figure 15 is interesting for a number of reasons. First, there's the Atom XML, which distinguishes our order from all the other orders in the feed. Then there's the order itself, containing all the information our barista needs to make our coffee – including our all-important extra shot! Inside the order entry, there's a link element that declares the edit URI for the entry. The edit URI links to an order resource that is editable via HTTP. (The address of the editable resource in this case happens to be the same address as the order resource itself, but it need not be.)

When a barista wants to change the state of the resource so that our order can no longer be changed, they interact with it via the edit URI. Specifically they PUT a revised version of the resource state to the edit URI, as shown in Figure 16.

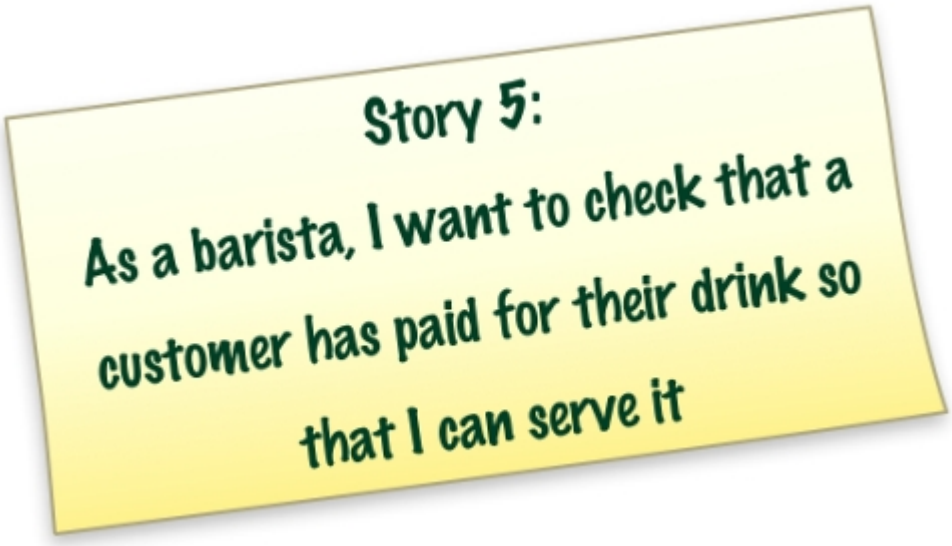
```
PUT /order/1234 HTTP 1.1
Host: starbucks.example.com
Content-Type: application/atom+xml
Content-Length: ...

<entry>
  ...
  <content type="text+xml">
    <order xmlns="http://starbucks.example.org/">
      <drink>latte</drink>
      <additions>shot</additions>
      <cost>4.00</cost>
      <status>preparing</status>
    </order>
    ...
  </content>
</entry>
```

Figure 16 Changing the order status via AtomPub

Once the server has processed the PUT request in Figure 16, it will reject anything other than GET requests to the /orders/1234 resource.

Now that the order is stable the barista can safely get on with making the coffee. Of course, the barista will need to know we've paid for the order before they release the coffee to us, so before handing the coffee over, the barista checks to make sure we've paid. In a real Starbucks, things are a little different: there are conventions, such as paying as you order, and other customers hanging around to make sure you don't run off with their drinks. But in our computerised version it's not much additional work to add this check, and so onto our penultimate story card:



The barista can easily check the payment status by GETting the payment resource using the payment URI in the order.

> In this instance the customer and barista know about the payment resource from the link embedded in the order representation. But sometimes it's useful to access resources via URI templates.
>
> URI templates are a description format for well-known URIs. The templates allow consumers to vary parts of a URI to access different resources.
>
> A URI template scheme underpins Amazon's S3 storage service. Stored artefacts are manipulated using the HTTP verbs on URIs created from this template: `http://s3.amazonaws.com/{bucket_name}/{key_name}`.
>
> It's easy to infer a similar scheme for payments in our model so that baristas (or other authorised Starbucks systems) can readily access each payment without having to navigate all orders: `http://starbucks.example.org/payment/order/{order_id}`
>
> URI templates form a contract with consumers, so service providers must take care to maintain them even as the service evolves. Because of this implicit coupling some Web integrators shy away from URI templates. Our advice is to use them only where inferable URIs

are useful and unlikely to change.

An alternative approach in our example would be to expose a /payments feed containing (non-inferable) links to each payment resource. The feed would only be available to authorised systems.

Ultimately it is up to the service designer to determine whether URI templates are a safe and useful shortcut through hypermedia. Our advice: use them sparingly!

Of course, not everyone is allowed to look at payments. We'd rather not let the more creative (and less upstanding) members of the coffee community check each-others' credit card details, so like any sensible Web system, we protect our sensitive resources by requiring authentication.

If an unauthenticated user or system tries to retrieve the details of a particular payment, the server will challenge them to provide credentials, as shown in Figure 17.

Request	**Response**
GET /payment/order/1234 HTTP 1.1 Host: starbucks.example.org	401 Unauthorized WWW-Authenticate: Digest realm="starbucks.example.org", qop="auth", nonce="ab656...", opaque="b6a9..."

Figure 17 Unauthorised access to a payment resource is challenged

The 401 status (with helpful authentication metadata) tells us we should try the request again, but this time provide appropriate credentials. Retrying with the right credentials (Figure 18), we retrieve the payment and compare it with the resource representing the total value of the order at http://starbucks.example.org/total/order/1234.

Request	Response
GET /payment/order/1234 HTTP 1.1 Host: starbucks.example.org Authorization: Digest username="barista joe" realm="starbucks.example.org“ nonce="..." uri="payment/order/1234" qop=auth nc=00000001 cnonce="..." reponse="..." opaque="..."	200 OK Content-Type: application/xml Content-Length: ... <payment xmlns="http://starbucks.example.org/"> <cardNo>123456789</cardNo> <expires>07/07</expires> <name>John Citizen</name> <amount>4.00</amount> </payment>

Figure 18 Authorised access to a payment resource

Once the barista has prepared and dispatched the coffee and collected payment, they'll want to remove the completed order from the list of outstanding drinks. As always we'll capture this as a

story:

Because each entry in our orders feed identifies an editable resource with its own URI, we can apply the HTTP verbs to each order resource individually. The barista simply DELETEs the resource referenced by the relevant entry to remove it from the list, as in Figure 19.

Request	**Response**
DELETE /order/1234 HTTP 1.1 Host: starbucks.example.org	200 OK

Figure 19 Removing a completed order

With the item DELETEd from the feed, a fresh GET of the feed returns a representation without the DELETEd resource. Assuming we have well behaved caches and have set the cache expiry metadata sensibly, trying to GET the order entry directly results in a 404 Not Found response.

You might have noticed that the Atom Publishing Protocol meets most of our needs for the Starbucks domain. If we'd exposed the /orders feed directly to customers, customers could have used AtomPub to publish drinks orders to the feed, and even change their orders over time.

Evolution: A fact of Life on the Web

Since our coffee shop is based around self-describing state machines, it's quite straightforward to evolve the workflows to meet changing business needs. For example Starbucks might choose to offer a free Internet promotion shortly after starting to serve coffee:

July – Our new Starbucks shop goes live offering the standard workflow with the state transitions and representations that we've explored throughout this article. Consumers are interacting with the

service with these formats and representations in mind.

August – Starbucks introduces a new representation for a free wireless promotion. Our coffee workflow will be updated to contain links providing state transitions to the offer. Thanks to the magic of URIs, the links may be to a 3rd party partner just as easily as they could be to an internal Starbucks resource

```
...
<next xmlns="http://example.org/state-machine"
    rel="http://wifi.example.org/free-offer"
    uri="http://wifi.example.com/free-offer/order/1234"
    type="application/xml"/>
...
```

Because the representations still include the original transitions, existing consumers can still reach their goal, though they may not be able to take advantage of the promotion because they have not been explicitly programmed for it.

September – Consumer applications and services are upgraded so that they can understand and use the free Internet promotion, and are instructed to follow such promotional transitions whenever they occur.

The key to successful evolution is for consumers of the service to anticipate change by default. Instead of binding directly to resources (e.g. via URI templates), at each step the service provides URIs to named resources with which the consumer can interact. Some of these named resources will not be understood and will be ignored; others will provide known state transitions that the consumer wants to make. Either way this scheme allows for graceful evolution of a service while maintaining compatibility with consumers.

The Technology you're about to enjoy is extremely hot

Handing over the coffee brings us to the end of the workflow. We've ordered, changed (or been unable to change) our order, paid and finally received our coffee. On the other side of the counter Starbucks has been equally busy taking payment and managing orders.

We were able to model all necessary interactions here using the Web. The Web allowed us to model some simple unhappy paths (e.g. not being able to change an in process order or one that's already been made) without us having to invent new exceptions or faults: HTTP provided everything we needed right out of the box. And even with the unhappy paths, clients were able to progress towards their goal.

The features HTTP provides might seem innocuous at first. But there is already worldwide agreement and deployment of this protocol, and every conceivable software agent and hardware device understands it to a degree. When we consider the balkanised adoption of other distributed computing technologies (such as WS-) we realise the remarkable success that HTTP has enjoyed,*

and the potential it releases for system-to-system integration.

The Web even helped non-functional aspects of the solution. Where we had transient failures, a shared understanding of the idempotent behaviour of verbs like GET, PUT and DELETE allowed safe retries; baked-in caching masked failures and aided crash recovery (through enhanced availability); and HTTPs and HTTP Authentication helped with our rudimentary security needs.

Although our problem domain was somewhat artificial, the techniques we've highlighted are just as applicable in traditional distributed computing scenarios. We won't pretend that the Web is simple (unless you are a genius), nor do we pretend that that it's a panacea (unless you are an unrelenting optimist or have caught REST religion), but the fact is that the Web is a robust framework for integrating systems at local, enterprise, and Internet scale.

Acknowledgements

The authors would like to thank Andrew Harrison of Cardiff University for the illuminating discussions around “conversation descriptions” on the Web.

About the Authors

Dr. Jim Webber is director of professional services for ThoughtWorks where he works on dependable distributed systems architecture for clients worldwide. Jim was formerly a senior researcher with the UK E-Science programme where he developed strategies for aligning Grid computing with Web Services practices and architectural patterns for dependable Service-Oriented computing and has extensive Web and Web Services architecture and development experience. As an architect with Hewlett-Packard, and later Arjuna Technologies, Jim was the lead developer on the industry's first Web Services Transaction solution. Jim is an active speaker and is invited to speak regularly at conferences across the globe. He is an active author and in addition to "Developing Enterprise Web Services - An Architect's Guide" he is working on a new book on Web-based integration. Jim holds a B.Sc. in Computing Science and Ph.D. in Parallel Computing both from the University of Newcastle upon Tyne. His blog is located at http://jim.webber.name .

Savas Parastatidis is a Software Philosopher, thinking about systems and software. He investigates the use of technology in eResearch and is particularly interested in Cloud Computing, knowledge representation and management, and social networking. He's currently with Microsoft Research's External Research team. Savas enjoys blogging at http://savas.parastatidis.name.

Ian Robinson helps clients create sustainable service-oriented capabilities that align business and IT from inception through to operation. He has written guidance for Microsoft on implementing service-oriented systems with Microsoft technologies, and has published articles on consumer-driven service contracts and their role in the software development lifecycle - most recently in The ThoughtWorks Anthology (Pragmatic Programmers, 2008) and elsewhere on InfoQ. He speaks regularly at conferences on subjects that include RESTful enterprise development and the test-driven

foundations of service-oriented delivery.

Link：http://www.infoq.com/articles/webber-rest-workflow

Related Contents：

- Interview and Book Excerpt: Thomas Erl's SOA Design Patterns
- Cloud Architectures Are Not Fully Thought Out Yet
- A Quick Look at Architectural Styles and Patterns
- Did you Perform a Silo Analysis as part of your SOA Implementation?
- Entity Services - Pattern or Anti-pattern?

Articles

Addressing Doubts about REST

Author: Stefan Tilkov

Invariably, learning about REST means that you'll end up wondering just how applicable the concept really is for your specific scenario. And given that you're probably used to entirely different architectural approaches, it's only natural that you start doubting whether REST, or rather RESTful HTTP, really works in practice, or simply breaks down once you go beyond introductory, "Hello, World"-level stuff. In this article, I will try to address 10 of the most common doubts people have about REST when they start exploring it, especially if they have a strong background in the architectural approach behind SOAP/WSDL-based Web services.

1. REST may be usable for CRUD, but not for "real" business logic

This is the most common reaction I see among people who are skeptical about REST benefits. After all, if all you have is create/read/update/delete, how can you possibly express more complicated application semantics? I have tried to address some of these concerns in the introductory article of this series, but this point definitely merits closer discussion.

First of all, the HTTP verbs - GET, PUT, POST, and DELETE - do not have a 1:1 mapping to the CRUD database operations. For example, both POST and PUT can be used to create new resources: they differ in that with PUT, it's the client that determines the resource's URI (which is then updated or created), whereas a POST is issued to a "collection" or "factory" resource and it's the server's task to assign a URI. But anyway, back to the question: how do you handle more complex business logic?

Any computation calc(a, b) that returns a result c can be transformed into a URI that identifies its result — e.g. x = calc(2,3) might become http://example.com/calculation?a=2&b=3. At first, this seems like a gross misuse of RESTful HTTP — aren't we supposed to use URIs to identify resources, not operations? Yes, but in fact this is what we do: http://example.com/sum?augend=2&addend=3 identifies a resource, namely the result of adding 2 and 3. And in this particular (obviously contrived) example, using a GET to retrieve the result might be a good idea — after all, this is cacheable, you can reference it, and computing it is probably safe and not very costly.

Of course in many, if not most cases, using a GET to compute something might be the wrong approach. Remember that GET is supposed to be a "safe" operation, i.e. the client does not accept any obligations (such as paying you for your services) or assume any responsibility, when all it does is

follow a link by issuing a GET. In many other cases, it's therefore more reasonable to provide input data to the server so that it can create a new resource via POST. In its response, the server can indicate the URI of the result (and possibly issue a redirect to take you there). The result is then re-usable, can be bookmarked, can be cached when it's retrieved … you can basically extend this model to any operation that yields a result — which is likely to be every one you can think of.

2. There is no formal contract/no description language

From RPC to CORBA, from DCOM to Web Services we're used to having an interface description that lists the operations, their names, and the types of their input and output parameters. How can REST possibly be usable without an interface description language?

There are three answers to this very frequently asked question.

First of all, if you decide to use RESTful HTTP together with XML — a very common choice — the whole world of XML schema languages, such as DTDs, XML Schema, RELAX NG or Schematron are still available to you. Arguably, 95% of what you usually describe using WSDL is not tied to WSDL at all, but rather concerned with the XML Schema complex types you define. The stuff WSDL adds on top is mostly concerned with operations and their names — and describing these becomes pretty boring with REST's uniform interface: After all, GET, PUT, POST and DELETE are all the operations you have. With regards to the use of XML Schema, this means that you can use your favorite data binding tool (if you happen to have one) to generate data binding code for your language of choice, even if you rely on a RESTful interface. (This is not an entirely complete answer, see below.)

Secondly, ask yourself what you need a description for. The most common — albeit not the only — use case for having some description is to generate stubs and skeletons for the interface you're describing. It is usually not documentation, since the description in e.g. WSDL format tells you nothing about the semantics of an operation — it just lists a name. You need some human-readable documentation anyway to know how to call it. In a typical REST approach, what you would provide is documentation in HTML format, possibly including direct links to your resources. Using the approach of having multiple representations, you might actually have self-documenting resources — just do an HTTP GET on a resource from your browser and get an HTML document containing data as well as a list of the operations (HTTP verbs) you can perform on it and the content types it accepts and delivers.

Finally, if you insist on using a description language for your RESTful service, you can either use the Web Application Description Language (WADL) or — within limitations — WSDL 2.0, which according to its authors is able to describe RESTful services, too. Neither WADL nor WSDL 2 are useful for describing hypermedia, though — and given that this is one of the core aspects of REST, I'm not at all sure they're sufficiently useful.

3. Who would actually want to expose so much of their application's implementation internals?

Another common concern is that resources are too low-level, i.e. an implementation detail one should not expose. After all, won't this put the burden of using the resources to achieve something meaningful on the client (the consumer)?

The short answer is: No. The implementation of a GET, PUT or any of the other methods on a resource can be just as simple or complicated as the implementation of a "service" or RPC operation. Applying REST design principles does not mean you have to expose individual items from your underlying data model — it just means that instead of exposing your business logic in an operation-centric way, you do so in a data-centric way.

A related concern is that not enabling direct access to resources will increase security. This is based on an old fallacy known as "security by obscurity", and one can argue that in fact it's the other way round: By hiding which individual resources you access in your application-specific protocol, you can no longer easily use the infrastructure to protect them. By assigning individual URIs to meaningful resources, you can e.g. use Apache's security rules (as well as rewriting logic, logging, statistics etc.) to work differently for different resources. By making these explicit, you don't decrease, you increase your security.

4. REST works with HTTP only, it's not transport protocol independent

First of all, HTTP is most emphatically *not* a transport protocol, but an application protocol. It uses TCP as the underlying transport, but it has semantics that go beyond it (otherwise it would be of little use). Using HTTP as a mere transport is abusing it.

Secondly, abstraction is not always a good idea. Web services take the approach of trying to hide many very different technologies under a single abstraction layer — but abstractions tend to leak. For example, there is a huge difference between sending a message via JMS or as an HTTP request. Trying to dumb widely different options down to their least common denominator serves no-one. An analogy would be to create a common abstraction that hides a relational database and a file system under a common API. Of course this is doable, but as soon as you address aspects such as querying, the abstraction turns into a problem.

Finally, as Mark Baker once coined: "Protocol independence is a bug, not a feature". While this may seem strange at first, you need to consider that true protocol independence is impossible to achieve — you can only decide to depend on a different protocol that may or may not be on a different level. Depending on a widely accepted, officially standardized protocol such as HTTP is not really a problem. This is especially true if it is much more wide-spread and supported than the abstraction that tries to replace it.

5. There is no practical, clear & consistent guidance on how to design RESTful applications

There are many aspects of RESTful design where there are no "official" best practices, no standard way on how to solve a particular problem using HTTP in a way conforming to the REST principles. There is little doubt that things could be better. Still, REST embodies many more application concepts than WSDL/SOAP-based web services. In other words: while this criticism has a lot of value to it, it's far more relevant for the alternatives (which basically offer you no guidance at all).

Occasionally, this doubt comes up in the form of "even the REST experts can't agree how to do it". In general, that's not true — for example, I tend to believe that the core concepts I described here a few weeks ago haven't been (nor will they be) disputed by any member of the REST community (if we can assume there is such a thing), not because it's a particularly great article, but simply because there is a lot of common understanding once people have learned a little more than the basics. If you have any chance to try out an experiment, try whether it's easier to get five SOA proponents to agree on anything than trying to get five REST proponents to do so. Based on past experience and long participation in several SOA and REST discussion groups, I'd tend to bet my money on the REST folks.

6. REST does not support transactions

The term "transaction" is quite overloaded, but in general, when people talk about transactions, they refer to the ACID variety found in databases. In an SOA environment — whether based on web services or HTTP only — each service (or system, or web app) implementation is still likely to interact with a database that supports transactions: no big change here, except you're likely to create the transaction explicitly yourself (unless your service runs in an EJB container or another environment that handles the transaction creation for you). The same is true if you interact with more than one resource.

Things start to differ once you combine (or compose, if you prefer) transactions into a larger unit. In a Web services environment, there is at least an option to make things behave similarly to what people are used to from 2PC scenarios as supported e.g. in a Java EE environment: WS-Atomic Transaction (WS-AT), which is part of the WS-Coordination family of standards. Essentially, WS-AT implements something very similar or equal to the 2PC protocol specified by XA. This means that your transaction context will be propagated using SOAP headers, and your implementation will take care of ensuring the resource managers hook into an existing transaction. Essentially, the same model in EJB developer is used to — your distributed transaction behaves just as atomically as a local one.

There are lots of things to say about, or rather against, atomic transactions in an SOA environment:

- Loose coupling and transactions, especially those of the ACID variety, simply don't match. The

very fact that you are co-ordinating a commit across multiple independent systems creates a pretty tight coupling between them.

- Being able to do this co-ordination requires central control over all of the services — it's very unlikely, probably impossible to run a 2PC transaction across company boundaries
- The infrastructure required to support this is usually quite expensive and complicated.

For the most part, the need for ACID transactions in a SOA or REST environment is actually a design smell — you've likely modeled your services or resources the wrong way. Of course, atomic transactions are just one type of transaction — there are extended transaction models that might be a better match for loosely-coupled systems. They haven't seen much adoption yet, though — not even in the Web services camp.

7. REST is unreliable

It's often pointed out that there is no equivalent to WS-ReliableMessaging for RESTful HTTP, and many conclude that because of this, it can't be applied where reliability is an issue (which translates to pretty much every system that has any relevance in business scenarios). But very often what you want is not necessarily some infrastructure component that handles message delivery; rather, you need to *know* whether a message has been delivered or not.

Typically, receiving a response message — such as a simple 200 OK in case of HTTP — means that you know your communication partner has received the request. Problems occur when you don't receive a response: You don't know whether your request has never reached the others side, or whether it has been received (resulting in some processing) and it's the response message that got lost.

The simplest way to ensure the request message reaches the other side is to re-send it, which is of course only possible if the receiver can handle duplicates (e.g. by ignoring them). This capability is called idempotency. HTTP guarantees that GET, PUT and DELETE are idempotent — and if your application is implemented correctly, a client can simply re-issue any of those requests if it hasn't received a response. A POST message is not idempotent, though — at least there are no guarantees in the HTTP spec that say it is. You are left with a number of options: You can either switch to using PUT (if your semantics can be mapped to it), use a common best practice described by Joe Gregorio, or adopt any of the existing proposals that aim to standardize this (such as Mark Nottingham's POE, Yaron Goland's SOA-Rity, or Bill de hÓra's HTTPLR).

Personally, I prefer the best-practice approach — i.e., turn the reliability problem into an application design aspect, but opinions on this differ quite a bit.

While any of these solutions address a good part of the reliability challenge, there is nothing — or at least, nothing that I'm aware of — that would support delivery guarantees such as in-order delivery for a sequence of HTTP requests and responses. It might be worth pointing out, though, that many existing SOAP/WSDL scenarios get by without WS-Reliable Messaging or any of its numerous predecessors, too.

8. No pub/sub support

REST is fundamentally based on a client-server model, and HTTP always refers to a client and a server as the endpoints of communication. A client interacts with a server by sending requests and receiving responses. In a pub/sub model, an interested party subscribes to a particular category of information and gets notified each time something new appears. How could pub/sub be supported in a RESTful HTTP environment?

We don't have to look far to see a perfect example of this: it's called syndication, and RSS and Atom Syndication are examples of it. A client queries for new information by issuing an HTTP against a resource that represents the collection of changes, e.g. for a particular category or time interval. This would be extremely inefficient, but isn't, because GET is the most optimized operation on the Web. In fact, you can easily imagine that a popular weblog server would have scale up much more if it had to actively notify each subscribed client individually about each change. Notification by polling scales extremely well.

You can extend the syndication model to your application resources — e.g., offer an Atom feed for changes to customer resources, or an audit trail of bookings. In addition to being able to satisfy a basically unlimited number of subscribing applications, you can also view these feeds in a feed reader, similarly to viewing a resource's HTML representation in your browser.

Of course, this is not a suitable answer for some scenarios. For example, soft realtime requirements might rule this option out, and another technology might be more appropriate. But in many cases, the mixture of loose coupling, scalability and notification enabled by the syndication model is an excellent fit.

9. No asynchronous interactions

Given HTTP's request/response model, how can one achieve asynchronous communication? Again, we have to be aware that there are multiple things people mean when they talk about asynchronicity. Some refer to the programming model, which can be blocking or non-blocking independently of the wire interactions. This is not our concern here. But how do you deliver a request from a client (consumer) to the server (provider) where the processing might take a few hours? How does the consumer get to know the processing is done?

HTTP has a specific response code, 202 (Accepted), the meaning of which is defined as "The request has been accepted for processing, but the processing has not been completed." This is obviously exactly what we're looking for. Regarding the result, there are multiple options: The server can return a URI of a resource which the client can GET to access the result (although if it has been created specifically due to this request, a 201 Created would probably be better). Or the client can include a URI that it expects the server to POST the result to once it's done.

10. Lack of tools

Finally, people often complain about the lack of tools available to support RESTful HTTP development. As indicated in item #2, this is not really true for the data aspect — you can use all of the data binding and other data APIs you are used to, as this is a concern that's orthogonal to the number of methods and the means of invoking them. Regarding plain HTTP and URI support, absolutely every programming language, framework and toolkit on the planet supports them out of the box. Finally, vendors are coming up with more and more (supposedly) easier and better support for RESTful HTTP development in their frameworks, e.g. Sun with JAX-RS (JSR 311) or Microsoft with the REST support in .NET 3.5 or the ADO.NET Data Services Framework.

Conclusion

So: Is REST, and its most common implementation, HTTP, perfect? Of course not. Nothing is perfect, definitely not for every scenario, and most of the time not even for a single scenario. I've completely ignored a number of very reasonable problem areas that require more complicated answers, for example message-based security, partial updates and batch processing, and I solemnly promise to address these in a future installment. I still hope I could address some of the doubts you have — and if I've missed the most important ones, you know what the comments are for.

Link: http://www.infoq.com/articles/tilkov-rest-doubts

Related Contents：

- REST and transactions?
- Financial Transaction Exchange at BetFair.com
- Should developers write their own transaction coordination logic?
- File System Transactions - still a problem area?
- Eric Newcomer On Difference Between RESTful vs. Web Service Transactions

Articles

REST Anti-Patterns

Author: Stefan Tilkov

When people start trying out REST, they usually start looking around for examples – and not only find a lot of examples that claim to be "RESTful", or are labeled as a "REST API", but also dig up a lot of discussions about why a specific service that claims to do REST actually fails to do so.

Why does this happen? HTTP is nothing new, but it has been applied in a wide variety of ways. Some of them were in line with the ideas the Web's designers had in mind, but many were not. Applying REST principles to your HTTP applications, whether you build them for human consumption, for use by another program, or both, means that you do the exact opposite: You try to use the Web "correctly", or if you object to the idea that one is "right" and one is "wrong": in a RESTful way. For many, this is indeed a very new approach.

The usual standard disclaimer applies: REST, the Web, and HTTP are not the same thing; REST could be implemented with many different technologies, and HTTP is just one concrete architecture that happens to follow the REST architectural style. So I should actually be careful to distinguish "REST" from "RESTful HTTP". I'm not, so let's just assume the two are the same for the remainder of this article.

As with any new approach, it helps to be aware of some common patterns. In the first two articles of this series, I've tried to outline some basic ones – such as the concept of collection resources, the mapping of calculation results to resources in their own right, or the use of syndication to model events. A future article will expand on these and other patterns. For this one, though, I want to focus on anti-patterns – typical examples of attempted RESTful HTTP usage that create problems and show that someone has attempted, but failed, to adopt REST ideas.

Let's start with a quick list of anti-patterns I've managed to come up with:

1. Tunneling everything through GET
2. Tunneling everything through POST
3. Ignoring caching
4. Ignoring response codes
5. Misusing cookies

6. Forgetting hypermedia
7. Ignoring MIME types
8. Breaking self-descriptiveness

Let's go through each of them in detail.

Tunneling everything through GET

To many people, REST simply means using HTTP to expose some application functionality. The fundamental and most important operation (strictly speaking, "verb" or "method" would be a better term) is an HTTP GET. A GET should retrieve a representation of a resource identified by a URI, but many, if not all existing HTTP libraries and server programming APIs make it extremely easy to view the URI not as a resource identifier, but as a convenient means to encode parameters. This leads to URIs like the following:

```
http://example.com/some-api?method=deleteCustomer&id=1234
```

The characters that make up a URI do not, in fact, tell you anything about the "RESTfulness" of a given system, but in this particular case, we can guess the GET will not be "safe": The caller will likely be held responsible for the outcome (the deletion of a customer), although the spec says that GET is the wrong method to use for such cases.

The only thing in favor of this approach is that it's very easy to program, and trivial to test from a browser – after all, you just need to paste a URI into your address bar, tweak some "parameters", and off you go. The main problems with this anti-patterns are:

1. Resources are not identified by URIs; rather, URIs are used to encode operations and their parameters
2. The HTTP method does not necessarily match the semantics
3. Such links are usually not intended to be bookmarked
4. There is a risk that "crawlers" (e.g. from search engines such as Google) cause unintended side effects

Note that APIs that follow this anti-pattern might actually end up being accidentally restful. Here is an example:

```
http://example.com/some-api?method=findCustomer&id=1234
```

Is this a URI that identifies an operation and its parameters, or does it identify a resource? You could argue both cases: This might be a perfectly valid, bookmarkable URI; doing a GET on it might be "safe"; it might respond with different formats according to the Accept header, and support sophisticated caching. In many cases, this will be unintentional. Often, APIs start this way, exposing a "read" interface, but when developers start adding "write" functionality, you find out that the

illusion breaks (it's unlikely an update to a customer would occur via a PUT to this URI – the developer would probably create a new one).

Tunneling everything through POST

This anti-pattern is very similar to the first one, only that this time, the POST HTTP method is used. POST carries an entity body, not just a URI. A typical scenario uses a single URI to POST to, and varying messages to express differing intents. This is actually what SOAP 1.1 web services do when HTTP is used as a "transport protocol": It's actually the SOAP message, possibly including some WS-Addressing SOAP headers, that determines what happens.

One could argue that tunneling everything through POST shares all of the problems of the GET variant, it's just a little harder to use and cannot explore caching (not even accidentally), nor support bookmarking. It actually doesn't end up violating any REST principles so much – it simply ignores them.

Ignoring caching

Even if you use the verbs as they are intended to be used, you can still easily ruin caching opportunities. The easiest way to do so is by simply including a header such as this one in your HTTP response:

```
Cache-control: no-cache
```

Doing so will simply prevent caches from caching anything. Of course this *may* be what you intend to do, but more often than not it's just a default setting that's specified in your web framework. However, supporting efficient caching and re-validation is one of the key benefits of using RESTful HTTP. Sam Ruby suggests that a key question to ask when assessing somethings RESTfulness is "do you support ETags"? (ETags are a mechanism introduced in HTTP 1.1 to allow a client to validate whether a cached representation is still valid, by means of a cryptographic checksum). The easiest way to generate correct headers is to delegate this task to a piece of infrastructure that "knows" how to do this correctly – for example, by generating a file in a directory served by a Web server such as Apache HTTPD.

Of course there's a client side to this, too: when you implement a programmatic client for a RESTful service, you should actually exploit the caching capabilities that are available, and not unnecessarily retrieve a representation again. For example, the server might have sent the information that the representation is to be considered "fresh" for 600 seconds after a first retrieval (e.g. because a back-end system is polled only every 30 minutes). There is absolutely no point in repeatedly requesting the same information in a shorter period. Similarly to the server side of things, going with a proxy cache such as Squid on the client side might be a better option than building this logic yourself.

Caching in HTTP is powerful and complex; for a very good guide, turn to Mark Nottingham's Cache Tutorial.

Ignoring status codes

Unknown to many Web developers, HTTP has a very rich set of application-level status codes for dealing with different scenarios. Most of us are familiar with 200 ("OK"), 404 ("Not found"), and 500 ("Internal server error"). But there are many more, and using them correctly means that clients and servers can communicate on a semantically richer level.

For example, a 201 ("Created") response code signals that a new resource has been created, the URI of which can be found in a Location header in the response. A 409 ("Conflict") informs the client that there is a conflict, e.g. when a PUT is used with data based on an older version of a resource. A 412 ("Precondition Failed") says that the server couldn't meet the client's expectations.

Another aspect of using status codes correctly affects the client: The status codes in different classes (e.g. all in the 2xx range, all in the 5xx range) are supposed to be treated according to a common overall approach – e.g. a client should treat *all* 2xx codes as success indicators, even if it hasn't been coded to handle the specific code that has been returned.

Many applications that claim to be RESTful return only 200 or 500, or even 200 only (with a failure text contained in the response body – again, see SOAP). If you want, you can call this "tunneling errors through status code 200", but whatever you consider to be the right term: if you don't exploit the rich application semantics of HTTP's status codes, you're missing an opportunity for increased re-use, better interoperability, and looser coupling.

Misusing cookies

Using cookies to propagate a key to some server-side session state is another REST anti-pattern.

Cookies are a sure sign that something is not RESTful. Right? No; not necessarily. One of the key ideas of REST is statelessness – not in the sense that a server can not store any data: it's fine if there is resource state, or client state. It's session state that is disallowed due to scalability, reliability and coupling reasons. The most typical use of cookies is to store a key that links to some server-side data structure that is kept in memory. This means that the cookie, which the browser passes along with each request, is used to establish conversational, or session, state.

If a cookie is used to store some information, such as an authentication token, that the server can validate without reliance on session state, cookies are perfectly RESTful – with one caveat: They shouldn't be used to encode information that can be transferred by other, more standardized means (e.g. in the URI, some standard header or – in rare cases – in the message body). For example, it's preferable to use HTTP authentication from a RESTful HTTP point of view.

Forgetting hypermedia

The first REST idea that's hard to accept is the standard set of methods. REST theory doesn't specify which methods make up the standard set, it just says there should be a limited set that is applicable to all resources. HTTP fixes them at GET, PUT, POST and DELETE (primarily, at least), and casting all of your application semantics into just these four verbs takes some getting used to. But once you've done that, people start using a subset of what actually makes up REST – a sort of Web-based CRUD (Create, Read, Update, Delete) architecture. Applications that expose this anti-pattern are not really "unRESTful" (if there even is such a thing), they just fail to exploit another of REST's core concepts: hypermedia as the engine of application state.

Hypermedia, the concept of linking things together, is what makes the Web a web – a connected set of resources, where applications move from one state to the next by following links. That might sound a little esoteric, but in fact there are some valid reasons for following this principle.

The first indicator of the "Forgetting hypermedia" anti-pattern is the absence of links in representations. There is often a recipe for constructing URIs on the client side, but the client never follows links because the server simply doesn't send any. A slightly better variant uses a mixture of URI construction and link following, where links typically represent relations in the underlying data model. But ideally, a client should have to know a single URI only; everything else – individual URIs, as well as recipes for constructing them e.g. in case of queries – should be communicated via hypermedia, as links within resource representations. A good example is the Atom Publishing Protocol with its notion of *service documents*, which offer named elements for each collection within the domain that it describes. Finally, the possible state transitions the application can go through should be communicated dynamically, and the client should be able to follow them with as little before-hand knowledge of them as possible. A good example of this is HTML, which contains enough information for the browser to offer a fully dynamic interface to the user.

I considered adding "human readable URIs" as another anti-pattern. I did not, because I like readable and "hackable" URIs as much as anybody. But when someone starts with REST, they often waste endless hours in discussions about the "correct" URI design, but totally forget the hypermedia aspect. So my advice would be to limit the time you spend on finding the perfect URI design (after all, their just strings), and invest some of that energy into finding good places to provide links within your representations.

Ignoring MIME types

HTTP's notion of content negotiation allows a client to retrieve different representations of resources based on its needs. For example, a resource might have a representation in different formats such as XML, JSON, or YAML, for consumption by consumers implemented in Java, JavaScript, and Ruby respectively. Or there might be a "machine-readable" format such as XML in addition to a PDF or JPEG version for humans. Or it might support both the v1.1 and the v1.2 versions of some custom

representation format. In any case, while there may be good reasons for having one representation format only, it's often an indication of another missed opportunity.

It's probably obvious that the more unforeseen clients are able to (re-)use a service, the better. For this reason, it's much better to rely on existing, pre-defined, widely-known formats than to invent proprietary ones – an argument that leads to the last anti-pattern addressed in this article.

Breaking self-descriptiveness

This anti-pattern is so common that it's visible in almost every REST application, even in those created by those who call themselves "RESTafarians" – myself included: breaking the constraint of self-descriptiveness (which is an ideal that has less to do with AI science fiction than one might think at first glance). Ideally, a *message* – an HTTP request or HTTP response, including headers and the body – should contain enough information for any generic client, server or intermediary to be able to process it. For example, when your browser retrieves some protected resource's PDF representation, you can see how all of the existing agreements in terms of standards kick in: some HTTP authentication exchange takes place, there might be some caching and/or revalidation, the content-type header sent by the server ("application/pdf") triggers the startup of the PDF viewer registered on your system, and finally you can read the PDF on your screen. Any other user in the world could use his or her own infrastructure to perform the same request. If the server developer adds another content type, any of the server's clients (or service's consumers) just need to make sure they have the appropriate viewer installed.

Every time you invent your own headers, formats, or protocols you break the self-descriptiveness constraint to a certain degree. If you want to take an extreme position, anything not being standardized by an official standards body breaks this constraint, and can be considered a case of this anti-pattern. In practice, you strive for following standards as much as possible, and accept that some convention might only apply in a smaller domain (e.g. your service and the clients specifically developed against it).

Summary

Ever since the "Gang of Four" published their book, which kick-started the patterns movement, many people misunderstood it and tried to apply as many patterns as possible – a notion that has been ridiculed for equally as long. Patterns should be applied if, and only if, they match the context. Similarly, one could religiously try to avoid all of the anti-patterns in any given domain. In many cases, there are good reasons for violating any rule, or in REST terminology: relax any particular constraint. It's fine to do so – but it's useful to be aware of the fact, and then make a more informed decision.

Hopefully, this article helps you to avoid some of the most common pitfalls when starting your first REST projects.

Many thanks to Javier Botana and Burkhard Neppert for feedback on a draft of this article.

Link: http://www.infoq.com/articles/rest-anti-patterns

Related Contents：

- What Makes Good REST?
- Debate: Should Architecture Rewrite be Avoided?
- Decisions driven by productivity concerns: Reasons, implications and limitations
- Preserving flexibility while using Active Record pattern
- Can architecture create a gap between developers and software they build?

Interview

Ian Robinson discusses REST, WS-* and Implementing an SOA

In this interview from QCon San Francisco 2008, Ian Robinson discusses REST vs. WS-*, REST contracts, WADL, how to approach company-wide SOA initiatives, how an SOA changes a company, SOA and Agile, tool support for REST, reuse and foreseeing client needs, versioning and the future of REST-based services in enterprise SOA development.

Ian Robinson is a Principal Consultant with ThoughtWorks, where he specializes in the design and delivery of service-oriented and distributed systems. He has written guidance for Microsoft on implementing integration patterns with Microsoft technologies, and has published articles on business-oriented development methodologies and distributed systems design.

InfoQ: Hi. My name is Ryan Slobojan and I am here with Ian Robinson. Ian, what do you currently consider to be the best technical option for creating a service-oriented architecture? WS-* or REST?

Ian Robinson: I think it is always going to depend; we are always going to have heterogeneous environments within the enterprise. There are likely technologies that are already in place, applications that are already in place that use WS-*, and it is unlikely that we would want to replace those just to impose some kind of uniform solution. A lot of the stacks offer a kind of homogenous development environment. And if we are developing the internals of an application or the internals of a service we can certainly take advantage of a lot of those WS-* compliant applications and interfaces. I think once we are looking for tremendous reach and scalability, when we are looking to extend across organizational boundaries, then we might want to look at more RESTful solutions. We will have technologies at either end, technology stacks that have simple HTTP clients, we can take advantage of those, we are not having to worry so much about incompatibilities between different versions of a WS-* specification. So I think really being able to take advantage of some of the web's infrastructure, some of the scalability that is inherent in that infrastructure might guide us towards adopting a RESTful solution for those parts of our SOA.

InfoQ: One of the other questions which comes to mind is how do you view the notion of contracts within a REST scenario?

Ian Robinson: I think contracts, as we are used to them from the web services stack, aren't necessarily as applicable in a RESTful environment or a RESTful solution. Nonetheless contracts are there, are present in one form or another, and it's probably worth investigating those in detail. But

first I will just talk a little bit about those web services contracts: WSDL and WS-Policy. Together they are typically said to comprise a web service contract. WSDL exposes the endpoints and the operations that can be performed at those endpoints. And WS-Policy asserts some of the quality-of-service characteristics that might be associated with that service.

WSDL in particular seems to encourage a remote-operation view of the world. It's very static, very upfront we are kind of committing very early to the way in which we want to consume a service. In a RESTful solution typically what we are trying to do is guide a client towards its goal. So the client makes a request of our service and we'll serve up some kind of representation, it might be a representation of an order for example. It may be that we want to progress that order through several states, ideally what we want to do inside that representation is advertise some of the possible transitions to the next stage or the next state in processing that order. So we're not necessarily having to advertise up front in some external contract what is that we can do with an order, the representation itself offers up several opportunities to manipulate that order. So this is that "hypermedia as the lever to application state" constraint within REST.

So really the contracts are being exposed gradually in a rather more dynamic fashion. We are still treating with the client over the course of several requests and responses. But we are enabling that client to make decisions on the fly as to where it wants to go next. One of the things that I think we really want to try and adhere to here is careful use of media types or MIME types. So a handful of good media types within their processing model, basically advertise what is a hyperlink, what are those hypermedia levers that are available within that particular media type.

So if we are building rather generic clients that can handle these particular media types, then they can begin to identify those possible state transitions, those possible on-the-fly elements of the contract and begin to act on them there and then. So we can start layering some application-specific intelligence on top of some very generic REST clients and we are using media types to guide those clients together with the particular representations that we are serving up. We are using those to guide those clients towards the successful completion of their goals. Does that make sense?

InfoQ: Yes, it does. So what are your thoughts on WADL, Web Application Descriptor Language?

Ian Robinson: We could use WADL. WADL effectively allows us to describe some of the operations that we could perform against any particular resource. We could use that in a WSDL-like way to provide some static upfront description of the service. Or we could actually use WADL to annotate some of those representations as they come back and say "Look here is today's contract for this order. Here are the things that you can do with this order today with this particular representation".

We have actually used WADL to annotate that representation and provide some kind of contract-like semantics on the fly. I think that is a preferable solution. Obviously we've talked about other media types, things like RDF has a rich processing model that allows us again to identify particular links and the semantics attached to those links so that we could interrogate a representation and then begin to progress it, in making further requests of the service.

InfoQ: How would you approach a large-scale company-wide SOA project?

Ian Robinson: An approach we have used successfully in the past at Thoughtworks, this is an approach I have used with a number of companies where they often have a very successful pedigree in mainframe application development and in other more recent kinds of application development as well, but they don't necessarily have any experience at SOA. Nonetheless there are very strong pressures for some significant change to take place, market conditions are changing, they are finding that their existing applications are difficult to change, expensive to change so all of this leads them to believe that they need some kind of SOA solution.

There's an obvious need there; they have a lot of experience in other kinds of application development; and they are aware that there's a lot of SOA experience out there but they don't necessarily posses it at this point in time. What do they do? How do they get started? What we are trying to do is provide some kind of accelerated route towards identifying and developing services, and then delivering them into the enterprise so as to deliver very real significant business value, value that meets that company's strategic goals. So typically what we do is present a very very simple map, ideally I would like to be able to draw that up for you, but it basically progresses from left to right and on the left hand side I just draw few boxes that represent some of those organizational units, their key goals, what is it they are trying to achieve, what's impeding them... In the middle of our map we diagram capabilities, and then over on the right-hand side we describe services and then specific technical implementations of those services.

And I say "What we are going to do together over the course, say, of a couple of months is we are going to begin populating this map with detail, we are going to start attaching very specific artifacts to parts of this map". So we are going to start trying to understand who some of the key stakeholders here are, what are their key goals, what are they trying to achieve, what's motivating them? And we are going to do that through a number of workshop exercises, things like that. And from that we are then going to start trying to identify some key or core capabilities that belong to your organization. What kind of capabilities or what abilities do you need to be able to furnish in order to meet some of those key goals? So again you want to draw those out as quickly as possible, doing just enough work to get some kind of sense of the overall scope of the engagement. Then we are going to start trying to assign those capabilities to services.

We say that services are hosts for one or more business capabilities. And then we can begin making some decisions about how we actually want to implement those services, what specific technologies, what architectural approaches do we want to take, in order to implement those services. So boiled down to its core, I talk about stories, capabilities, services and contracts. Stories I take from things like Behaviour-Driven Development, and using stories and the story format to describe a role -- "I want to achieve this, so that" -- and then I describe some kind of value attached to achieving that goal, so identifying a role, a goal and a value attached to that goal.

We use that kind of story format very often with people with strategic responsibilities at the beginning of an initiative. We'll ask them really to try and describe some of their key business goals, and the value that they attach to them. And then, as a complement to that, we try and derive what sort of capabilities are that that company possesses or needs to furnish in order to meet those goals. So what we are trying to create is a capability map, very simple, often this kind of hierarchical description, but it's a description really of what it is that the company does or what it is trying to

achieve. And then that becomes the basis for a whole bunch of other conversations both with strategic stakeholders and some of the operational staff.

Which of these capabilities are core to your business? Which of them differentiate them, or differentiate you from your competitors? Which of them do you do well today, and which of them do you do badly? Are there any that you could outsource? And what are your kind of quality-of-service expectations around these capabilities? And we are not at this moment in time talking about how we implement them, but if you need to be able to source parts for second-hand cars, what's the turnaround time on that kind of expected provision of that service or that key capability? So we can ask all of these questions, we can begin to derive some kind of quality-of-service characteristics.

Then we are beginning to hone in on some of the key capabilities, the things that are very very significant, very important to this company, or things that they are not currently doing well but which nonetheless they ought to be doing better. We can then start to identify services, start assigning those capabilities to services. And then we can start making some decisions about how to actually build these services. Now often that takes place in an environment where there is a whole bunch of in-flight projects, so again as part of this very quick start way of approaching an SOA initiative, we start to try and map some of those key capabilities and perhaps some of the candidate services that we've identified to in-flight projects so we are beginning to create this shared understanding between several teams, several different groups of stakeholders, we are trying to bring them together within this very simple map, and help them understand the several dependencies between different project streams.

This is really an iterative exercise, so as I say we'll go through a number of workshops to identify some very high levels and key capabilities and we might immediately, following on from that, try to identify some candidate services and start trying to deliver very quickly some working software that helps satisfy some of those service behaviors. But we'll constantly be going back, engaging more stakeholders, drilling down and discussing more of those capabilities in detail. But again it's a conversation around what is it you are trying to achieve, why is it important to you, what kind of quality-of-service characteristics do you attach to these things? So this is the basis of the kind of conversation that can join up several different parties within the organization, some of those business stakeholders, some of those technical stakeholders. So the capabilities become this ubiquitous language and then the services and the service implementations are really implementation details. Now we can begin to share some of that with that group of stakeholders, but really I see the link as being really those conversations around the capabilities.

InfoQ: One of the questions that comes to mind is how does implementing a SOA change an organization which did not previously have one? How does it change the flow of work within an organization and capabilities?

Ian Robinson: I think one of the first effects really is a whole bunch of people who have not necessarily talked to one another too much actually coming together and creating a shared understanding of what it is they are trying to do, and why it's important, and doing it in some very simple practical terms. We are not using strictly technical language at this point in time. So we're creating this shared understanding, and often that's a real breath of fresh air for some of these

organizations where they have been locked in very siloed efforts, there is a lot of repeated or duplicated effort across the organization. We are beginning to identify some of that and we're helping people bridge some of those gaps.

As we start actually delivering working software, we are also trying to encourage those teams to collaborate, to identify dependencies and responsibilities. This team over here might have some very real responsibilities to your team over there, how can we communicate those and how can we continually enforce those responsibilities. And one of the ways that we like to do that is to actually share tests. So your team could actually create a suite of tests that assert some of the expectations that you have of my service -- this is the way in which you want to interact with it.

These are the parts of the service that are important to you. And you give us those suites of tests, and we might incorporate them into our continuous integration environment, so we have this very practical programmatic asserted behavior. So we are actually getting these programmatic contracts being exchanged between teams. We are also trying to encourage teams to be more long-lived, so they live with a service from its inception through to its operation rather than a core team being dedicated solely to development and then handing over to some other support function.

Now again that's often quite a significant organizational change, it's not always easily accomplished and it's not always appropriate, but trying to encourage teams to have this long overall duty of care to the lifetime of the system and to take account of the several different parties that are going to be responsible for that system and thinking about that a little earlier in the development lifecycle. So again there are those kinds of very practical changes that we begin to see take place within an organization as people are coming together around these shared goals and as we are thinking of very practical ways of communicating, exchanging understanding and creating these programmatic contracts between teams.

InfoQ: One of the thoughts which comes to mind while listening to what you are saying is that a lot of what you are describing sounds a lot like an Agile implementation. Do you see a good SOA architecture within an organization and Agile being necessarily intertwined?

Ian Robinson: Yes, but I want to caveat that quite a bit. There are three terms that I'm quite wary of, particularly around SOA initiatives: those terms are Agile, integration and business process. So, I think it's a kind of open question even today: can we create organizational or enterprise agility using Agile project management or software delivery methods? So people often say: can we do Agile SOA? Sometimes that's a CIO is very skeptical, saying the rigors of SOA don't seem to go with this, seeming gung-ho attitude that Agile has, how can the two come together? And sometimes it's a kind of dogmatic Agile practitioner worrying that the kind of protracted exercises that seem necessary to SOA can't be made Agile and therefore they are almost questioning SOA's right to exist.

So can we do Agile SOA? To me that's the wrong question. Better to ask: what can we do to better meet an organization's key strategic goals? Whatever is appropriate is appropriate. Now as Agile practitioners, we've got a whole bunch of activities, practices, principles that we can bring to the table. Some of them are more appropriate than others, some of them have to be modified. Because we are dealing often with teams at a distance, projects working within different time streams, it's very difficult to coordinate all of those things. We can't always just pour our knowledge out onto a

bunch of cards around a table, we are often dealing with a scale where that's not practical or possible.

Nonetheless, there are some key things that we take away from Agile, this desire to have insight, day-to-day or minute-to-minute insight into our systems, and the way they are behaving, desire to have very strict and close feedback loops. So I'm often looking for strategies and techniques and practices that can encourage those things and that's an attitude that I bring from Agile, but whether or not I am doing Agile SOA to me is an ecumenical matter, I am not overly interested whether or not I am doing Agile SOA. I said a couple of other terms that I am wary of as well, integration. So I often see organizations with some kind of integration group, or they are talking very often about doing integration.

Integration to me is a bit of a bad smell, it suggests that what you are trying to do after the fact is glue together a bunch of systems and fix up a bunch of bad decisions. So I prefer to talk about intrinsic interoperability, though I can barely say the word, and within a RESTful solution that is often leading to serendipitous reuse. But intrinsic interoperability, even worse the second time, over integration. Now I recognize that integration is absolutely necessary. There's a whole load of integration activity that takes place day-to-day and very often when we are building out a service a lot of the internal implementations of that service might be a bunch of integration activities. But let's not prize integration as something that we do successfully and it makes the world a better place.

And the last term that I am wary of, and again I recognize that it is necessary, but it's this term "business process". Business process to me, it seems an overly formal term, it's a term that is used by a bunch of specialists but it's not necessarily meaningful to those people on the ground who are actually achieving a company's goals day to day. There's a whole bunch of ad-hoc activities, collaborations that take place. If you are of a particular strategic mind, you might try and divine business processes or overlay this formality on top of them. But if we talk too quickly about business processes we end up talking about these rather fragile assemblies or sequences or workflows. And we say "Well this is the way the world is today and this is the way it ought to be".

In fact a business process again is often an implementation detail of one of those capabilities. Today we implement the capability to source second-hand parts for your car, some hurried staff in the call center have to go through a filing cabinet, that's the business process today. Tomorrow we might automate it, but they are implementation details. So again I recognize that it is a very important term, but if I am looking for an Agile SOA solution that is oriented around integration and business processes, I think I am slightly off track. If I am trying to deliver to some of those significant goals, providing solutions that are capable of evolving, where we've got rapid feedback and minute to minute insight into the behaviors of our system, that's good. If we are doing integration inside of our services that's absolutely necessary integration but protecting all the other consumers of that service from that kind of messy detail, that's good. And if we are talking about business processes as a pure implementation detail of some of those key capabilities, again that's fine.

InfoQ: Can you talk about the current state of tool support for developers who want to get going with REST-based web services and frameworks and tools and best practices?

Ian Robinson: Ok, so the state of tool support for doing RESTful development. Well, the base tools

are there in most development languages - HTTP clients and ways of hosting solutions so they are listening to an HTTP endpoint as well... Let me have a think about that. I am not overly familiar with a wide variety of tools, and I would say I suppose there isn't terrific tool support in a way that there is with the WS-* stack. So typically as a developer what we want when we are developing a web service, is some WSDL that we can then use to autogenerate some kind of proxy.

If some of the representations that we are serving up in a RESTful manner conform to some XML schema, then it's likely we have got some kind of automated tool support for producing those strongly-typed representations in our own language for serializing and deserializing. I don't think that's absolutely necessary, I much prefer to, again if we are using XML, to XPath out the parts of the message that I am really really interested in and discard all the rest. And I'd probably adopt that approach even when using the web services stack. So again my preference is not to autogenerate proxies and clients against a particular version of the schema, but instead just to parse out the bits of the message that I am really interested in, and I would adopt exactly the same approach in a RESTful solution.

What I have done a little work on recently is a very simple DSL for expressing a client's expectations with regard to a particular message or representation. So that DSL it looks a little like YAML basically a bunch of nested terms together with a type, I expect this to be a string, I expect this to be an integer. But it is very concise but then from that I can generate a whole bunch of things, I can generate a bunch of XPath assertions so that I can validate incoming messages if I want or I can validate them on the way out. I can generate serializers and deserializers that are dedicated to satisfying my expectations with regard to the kind of messages that you are producing. I can also use it to generate a graph that, with regard to this particular schema, this graph begins to describe some of the expectations that different clients have of that schema.

So I am building up effectively this kind of social networks for contracts. So all these different artifacts we can generate off the top of something produced with this DSL. It's something that I am playing with at the moment, but that seems to me to be a way of being able to express my expectations of a message and then create type classes that are really dedicated to those expectations, towards servicing those expectations. So I've kind of got a bit off track in terms of your question around REST tool support. In developing some solutions recently with Atom and AtomPub, what I've really wanted to ensure is that the protocol and the way in which those clients are interacting with those feeds is being adhered to, so I want to create a whole bunch of unit tests around the service that is generating those feeds.

And I want to be able to assert that specific HTTP headers are coming back, certain response codes in response to a particular stimulus. What I found, and I was doing this on the .Net framework, what I found was that you can very quickly get into that HTTP context, but for every test, what you are having to do is actually instantiate a service over HTTP and communicate with it. So what I have done is just create a simple wrapper around that HTTP context, it's an interface that I own, and then I can mock it out and obviously set expectations with regard to that mock. And then I've got a separate bunch of tests that just assert that specific implementations of that interface actually delegate to the .Net framework. One of the things that we want to be doing is testing the protocol; like I said, that's in terms of status codes, headers, that kind of stuff.

InfoQ: How can you achieve reuse, and how can a provider foresee the needs of its clients?

Ian Robinson: Going back to something that Martin said in the keynote this morning, our experience is that reuse often happens after the fact. Let's do one thing well -- we understand the specific context in which this particularly piece of functionality is to work, let's deliver to that. Then we begin to identify opportunities for reuse. If we have a suite of unit tests then we have a bedrock of asserted behavior that will allow us to evolve a piece of software towards a more generic solution. So as new use cases come along, we are identifying those opportunities for reuse, we can evolve our software quite rapidly and that suite of tests helps ensure that our existing obligations are being satisfied.

What we have done at Thoughtworks with a number of clients on a number of projects is then we'll extend this, and it's something that I talked about earlier in fact, whereby, if we are talking about reusing aspects of services, parts of services, then we'll often have client or consumer teams give us a suite of tests that describe their expectations with regard to our service or, if we are earlier in the software or the service development lifecycle, then together these teams will help establish obligations and expectations. You need this of me, you need to see this happen, how can we describe that as a contract between the two of us, and how can we then turn that contract into perhaps a suite of tests, perhaps again XPath or Schematron, that say "you send me this message or if I send you this message this is the kind of response that I expect".

And these are the parts of the response that I am particularly interested in. So, when you give me that suite of tests you are basically communicating to me your expectations and then it may be that another client or consumer comes along and gives me a similar suite of tests, and gradually I'm building up my overall aggregate set of obligations, so I'm beginning to learn. This often takes place within a controlled environment, within the enterprise, it's not necessarily an Internet-scale solution. But I'm beginning to understand what my obligations are with respect to several different parties and then if I want to evolve my service, well I'm free to do so just as long as I don't break any of those expectations.

On your side what you're promising is, you know, you've advertised an enormous schema - actually we are interested in these five different fields, what we're promising is that we are only going to consume those five fields and we are going to throw anything else away. So you're free to change everything else as long as you continue to provide those five fields, so the promise on your side is you are not importing or making use of stuff that you are not telling me about. And then as a service provider I'm quite free to change my schema to evolve it as long as I don't break any of those existing obligations and in fact I can make a change that is ostensibly a breaking change but as long as it is not actually breaking any of my extant clients, who cares? So the versioning... There's definitely versioning taking place here, but it is often at a slower pace.

I don't necessarily need to version if I know that I'm going to continue to satisfy all those existing clients. It's when I identify a real breaking change and I need to communicate that to you, we need to be able to identify some way of moving forwards. That might be that I provide an alternative implementation that supports you for the next 6 months, it might be that you have to begin to make some changes now.

InfoQ: You had mentioned versioning. What do you consider some of the best approaches with

regards to versioning?

Ian Robinson: I think it's firstly about providing a platform for evolution. So with XML schema we can provide for extensibility points, we can design schemas with extensibility in mind. That's often quite cumbersome, and the messages as they've evolved over time actually begin to look rather awkward and are not necessarily as expressive as they might be. I've talked a little around what we're calling consumer-driven contracts and the fact that they help me understand when a change is really a breaking change and when what's a seeming breaking change actually doesn't really disturb the universe at all. So, there are very real demands for a versioning strategy within an organization.

You know these things can often be very long lived, and we've seen mainframe applications that have lived for twenty years or more, it will be wonderful if the kinds of solutions that we are producing today could have a similar lifetime. It's almost inevitable that they are going to change and therefore we do need to start thinking about those versioning strategies. I don't think there is actually a great versioning story in a lot of tool sets today and in a lot of the frameworks and I think it is a problem that is beginning to make itself felt, and I think a lot of those technology stacks and a lot of solutions and the frameworks are beginning to address that. But I try and take a very cautious approach, basically having consumers only consume what's absolutely important to them, discard the rest, have them try to communicate some of those expectations to a provider, and that helps the provider understand when they are free to change, but at some point we do need to version, and that's the point where we might have to take advantage of some of those extensibility points that we have provided for, it might be that we have to provide a wholly new interface.

InfoQ: What do you see as the future of REST-based web services in enterprise SOA development?

Ian Robinson: I think we are learning today that a lot of the enterprise solutions that we have built in the past are very much confined to the enterprise, and we have often abused or completely disregarded some of the benefits that things such as HTTP and the larger web infrastructure have to offer us. We are also discovering today that a lot of the value that we want to generate within an organization is dependant upon its interactions and its collaborations with other organizations. So far more communication across organizational boundaries. Parts of the web services stack inhibit that kind of cross-organizational growth. We have a proliferation of specifications and often for a particular specification there are several different versions.

We are finding it increasingly difficult to get that kind of intrinsic interoperability across organizational boundaries using the web services stack. RESTful solutions can help us extend our reach in this regard. We are taking advantage of a constrained interface, but we are beginning to surface and describe a rich pool of resources and we are helping identify each of those resources and make them available to our clients and to other organizations. And we are helping guide those clients towards successful, the successful conclusion of their goals. So we talked about that earlier in terms of serving up representations that help a client achieve its goals and we are beginning to advertise what the next step in the process is.

Now, I think that we can learn from that even if we want to import some of those lessons into the way in which we are building solutions on top of the web services stack. Identifying resources in and of itself is a very useful exercise, so adopting kind of resource oriented thinking often help us identify

things which are significant to a company, which generate value on behalf of a company, give them a name. Often those things are otherwise buried away in some implementation detail - we're beginning to surface them, give them names, make them addressable. The idea that processes are not defined once and for all, that they might gradually evolve over the course of a long-lived conversation, interaction across organizational boundaries.

Again, how can we guide clients, how can we advertise what is possible to do today, we might be advertising something completely different tomorrow, we might be introducing for example some kind of advertising campaign in the midst of the ordinary process and if a client can recognize those additional elements of that process, that we're advertising on the fly, they might be able to take advantage of that. But clients that don't recognize it can still meet their core goal of generating an order. So, I think we are seeing solutions today that are emerging that are beginning to take advantage of some of this thinking, beginning to introduce some RESTful ideas across a broader range of solutions and that is the kind of influence that I would like to have in the next few years.

View Full Video：

http://www.infoq.com/interviews/robinson-rest-ws-soa-implementation

Related Contents：

- How Relevant Is Contract First Development Using Angle Brackets?
- REST – The Good, the Bad and the Ugly
- Quest for True SOA
- InfoQ Minibook: Composite Software Construction
- Presentation: Scott Davis on Real World Web Services

Interview

Jim Webber on "Guerilla SOA"

In this interview, recorded at QCon London, Jim Webber, ThoughtWorks SOA practice leader talks to Stefan Tilkov about Guerilla SOA, a lightweight approach to SOA that does not rely on big middleware products, a message-oriented architectural style called MEST and its differences to REST, and the SOAP Service Description Language (SSDL).

Dr. Jim Webber is the SOA Practice lead for ThoughtWorks, where he works on Web Services-based systems for clients worldwide. He has extensive Web Services architecture and development experience and was the lead developer with Hewlett-Packard on the industry's first Web Services Transaction solution. Jim is is co-author of the book "Developing Enterprise Web Services - An Architect's Guide."

InfoQ: This is Stefan Tilkov at QCon and I am interviewing Jim Webber. Can you tell us a bit about yourself?

Jim: I work for a small consulting organization called "ThoughtWorks", who you may have heard of, and I do a lot of SOA and Web services work for them, particularly with an emphasis on dependable systems. Maybe it's because I am a pessimist, but I look for those kinds of situations when things go wrong and figure out ways of mitigating that kind of risk.

InfoQ: So the title of your talk here at QCon is "Guerilla SOA". Can you tell us a little bit about what that is supposed to mean?

Jim: A lot of SOA projects I have seen, have been somewhat akin to mobilizing an army. You have hundreds of consultants, a whole bunch of armaments in the form of huge, sophisticated middleware platforms. The whole thing is very heavy-weight and cumbersome. I feel that when you are going for that kind of big, upfront SOA deployments, you lose a lot of opportunities to prioritize, to deliver business processes based on your business priorities and your business values. The Guerilla SOA aspect tries to turn that around a little, so we're looking for much more lightweight engagements, if you'd like in military terms. We want to address specific discrete business problems, organized by priorities according to the business stakeholder and get those processes implemented rapidly in an incremental way with lots of feedback. So we can actually start to prioritize across the business - which process is the most valuable, which ones are most heavily used and implement those first without having to wait for a big program of work to be established, to put the enterprise service bus in place or other kind of technical dependencies. So it is kind of almost a, tongue in cheek really, but a hit and run, deliver often, and incrementally kind of SOA option.

InfoQ: So is this something that is only usable in smaller scenarios or does it scale up to big

deployments or big scenarios?

Jim: The nice thing is that it works in either because you have specific priorities that the business gives you at any given time and you focus on those, and as long as the business can keep coming to you and saying "I now need that this process implemented" then you can scale up ad infinitum until the point where you automated all of the processes of a given domain in a given business, so it scales fairly well.

InfoQ: It sounds a little different than the approach prioritized by some of the vendors, I think. So is this approach compatible with large scale middleware products as well?

Jim: I think the approach helps us the to decouple dependencies between what the business people want and the tools we have available. So although I poke fun at the ESBs and so on, I would absolutely use those tools where it makes sense to me to implement the process that I have been instructed to automate. If it doesn't make sense then I won't use them and I will use other tools. I will use anything from simple Java apps right the way through to full message broker, store&forward-based architectures where it makes sense within my current context. But the important thing is that I don't let my current development context bleed. I don't let those abstractions leak into other development projects.We like to keep each process that we are implementing relatively isolated so that then the service ecosystem grows and can be reused. It has this emergent behavior that we never expected. On the other hand if we allow everything to bleed together in a big SOA platform you tend to get tight coupling and that restricts your options for evolution further down the line and it restricts your options for this kind of interesting emergent behavior, which me and you as geeks could but the business people couldn't see because they have this much broader view of processes as a whole.

InfoQ: So you mentioned tight coupling as a risk. Can you elaborate on that?

Jim: Sure. It is the classic scenario. If I've got two systems which are tightly bound, I change one I risk breaking the other. We saw this back in the day with CORBA applications where we tightly coupled through IDL and we see it today in Web services where we tightly couple through another IDL called WSDL. If we are sharing type systems and I want to change my type system in my program that can have a ripple-through effect which is going to hurt you. So when I come to you and say "I'm going to make changes" your first reaction is "No, because you're going to break me!" and then we get into this paralysis, where neither of us can make progress because we're so scared of damaging each other. Then you need strong governance and so on and someone to come with a strong arm and make both parties move. It's a reluctant high friction environment to be in and yet had we decided not to share technical abstractions at that level chances are that we'd be much freer to evolve and innovate locally without disturbing or breaking anyone else globally because the abstractions we use internally would be different to the abstractions we share with other services around the ecosystem.

InfoQ: So what would be an alternative to that approach?

Jim: An alternative to the approach of sharing type system, for example?

InfoQ: Yes you said that we actually have the same problems we had with CORBA but we now have

them with a different type of technology. Do you have an idea of what we could do instead?

Jim: Instead of sharing types, I think we should start to share business messages, or schemas for business messages, owned not by the technical people, but by the business people. That gives me as a developer of a service an interface which I can make sure that I adhere to and honor a contract in my service implementation. You can also see that contract in your service implementation and you can understand that you're going to get these kinds of messages in and out. The point being at the technical abstractions that you are using to implement that type, you may have some interesting class hierarchy, are never exposed, so I can never bind to them so we never get coupled at that level. The coupling we have is just on the messages we depend on.You look at my service's contract, you see the messages that come in and out at my service and somewhere deep in the bowels of my service, I kind of have ways of extracting the information and using it to do some processing and you so in your service. In between we have this very neutral integration domain, which is just the business messages as recognized by the business stakeholders. Another benefit of that is that the business stakeholders can tell you when you've got things right and when you've got things wrong, which is tremendously difficult if you are using lower level abstractions like the type system. Because the business guys know that this message used to get sent by fax from Sydney to London and they knew the semantics of that and if you can show them the same thing in your automated electronic workflows they can say: "Yes. That's right!" or perhaps even more valuable: "No you have got it wrong! Stop! Do it this way!" So you don't go off on a tangent building a solution to what you think is the problem, you build a solution to the actual problem.

InfoQ: So you mentioned services and messages as two abstractions? What about operations?

Jim: Operations are an abstraction which I do not believe exists in a service oriented architecture. They may well exist in your implementation of a service but that is nothing that I want to share with you. This is a technical detail which is my business inside my implementation. When I think about an SOA, I like to think about the notion of letter boxes. So all I can do is deliver a message to you and at some point you might open it, read it, think: "Yes, I understand what that message is." and then you will go away and process it - or not. If I send you a nonsense message you may be graceful enough to fault and tell me so, but literally we don't have any tight coupling in the form of an operation abstraction. I can't invoke you because for all I know you are in a 3rd party system in a different organization so I don't have that strength. You are not a local object to me, we don't have a call stack, I can't poke you. All I can do is request: "Could you possibly have a look at this message and maybe if it suits you do some processing on it", rather than the more tightly coupled operation abstraction.

InfoQ: So for this to work all of the information that the other party needs to process this information has to be within that message?

Jim: Absolutely. And this is from a style of architecture which we called MEST or Message Exchange, which was a deliberate paying of respect to REST where some of our inspiration came from, in so far as this letterbox is a uniform interface through which we poke messages. If we'd map it onto HTTP it would be a POST. You can also use SMTP SEND or whatever else you choose. The message would contain 2 things: it would contain the business payload which is effectively the purchase order, the invoice, those kinds of things that business people process and it would contain some metadata,

potentially contains some metadata or anyway, which sets the processing context for that payload. So it may set security context, it may set transaction context, that kind of thing. The MEST idea is that I'm delivering you a message; you are going to go away, set the context of processing that message, examine that message, find whether it makes sense, go away and process that message. End of story. At some point later a message comes into my letterbox, I open it and say: "Ok. That's from Stefan." And I know what this means.It's actually correlated somehow, typically with WS-Addressing RelatesTo and so on, with that message I sent him earlier. Now I can go into my implementation and finish the processing I was doing, which originally caused the message to be sent to you. And that's a really nice decoupled way of doing things. I'm not binding to you directly; the only things I'm binding to in a technical sense are messages which are in my stack, in my process space, which is very safe to bind to, whereas if we go back to the operation abstraction, if I'm bound to you and invoking and for some reason you're down because the network is down or you're in a different company and the firewall rules suddenly got restrictive suddenly I break, I get this horrible "Internet timed out" exception or something meaningless; whereas if I'm just treating messages going up and down in my stack as the things I used to cause processing or things that I created as a side effect of processing, it's actually a robust pattern for implementing individual services as well as a nice decoupled scalable pattern for building up service ecosystems.

InfoQ: As you mentioned REST, I just have to ask how do you compare the two, MEST and REST? Where are the commonalities and where are the differences?

Jim: Sure. Commonalities are pretty obvious: uniform interface, so REST has five operations each resource implements; in MEST every service has one interface which is effectively poke a message in here. Differences are MEST is very much more akin to traditional MOM; it's about passing messages over some transport, whereas REST uses the hypermedia engine. They are kins because they both aim for large scalable systems, but whereas RESTful systems tend to look like the web, MESTy systems tend to look like TCP. Just make connection, post the message, close connection, that kind of thing. So there are similarities and I think both models have been proven out. Tongue in cheek, I'd say TCP happens to be slightly bigger than the web so maybe the METS solution is more scalable, but I'm not to upset the REST jihadists at this point.

InfoQ: What you described actually seems to fit very nicely with the ideas behind SOAP and you also mentioned WS-Addressing. It just doesn't exactly seem to match WSDL.

Jim: Right. WSDL is an IDL. WSDL's abstractions are operations. It has some other drawbacks in so far as it's quite a verbose IDL. I think the difficulty comes when you start to get past "StockQuote" web services and you need to be able to have a conversation, a long-lived conversation with a service, which WSDL doesn't have the abstractions to support. The longest conversation you have with a WSDL-described service is requests/response. Some time ago this started to become quite a chafing limitation for me and some other guys, Savas Parastatidis and some of the guys working at CSIRO in Sydney, Australia; we decided we are going to do something about it. And this is when we wrote SSDL. SSDL has a spectrum of possibilities; at one end is just a less verbose replacement for WSDL 2, it's completely isomorphic to the capabilities that WSDL 2 gives you, and at the other end it's a superset on what's available in WS-CDL, WS-BPEL and WSDL.

So we are able to describe long-lived conversations between multiple web services in a structured way, in a way that we can verify that that conversation won't deadlock so we can put it through model checkers and so on; so we can actually get a whole lot of static analysis about how end-to-end systems are going to look and still support this notion of quite intricate conversations with a service. A typical message exchange pattern in SSDL might be two requests, followed by seven responses, followed by another request, an optional response, and three more requests. And we can build arbitrary conversation patterns in it, which is really good when you think that most web services are going to be used to host business processes and most business processes are workflows which have this kind of more chatty or conversational kind of interaction pattern which is really difficult to capture in WSDL being limited to requests and responses. So SSDL gives you this capability to describe workflows effectively which I think is going to be one of the sweet spots of the SOA web services going forward.

InfoQ: So is SSDL a standard?

Jim: No. SSDL was an effort by some academic researchers and practitioners to see what a contract metadata language would look like, if we were freed from the tyranny of the operation abstraction. Right now it's been in the community for a couple of years, it's got some pretty good feedback, a lot of the web services guys know about it and have commented favorably about it, but it doesn't have the backing of any of the large vendors, although some of the people involved in it now work for large vendors and large research organizations, there is nothing official. Our hope originally was maybe we can just inspire some thinking in the vendors that are providing tools, so the vendors can give us tools that do this workflowy type stuff. And that's happening, there has been some discussion in the community, but now to keep momentum going the community itself has started to develop tools.

InfoQ: Are there any implementations of SSDL yet?

Jim: Yes. When we first released SSDL Savas Parastatidis of Microsoft had a simple SSDL tool that would do some basic contract generation, validation and so on. But more recently Patrick Fornasier of University of New South Wales in Sydney has built a complete SSDL stack on top of Windows communication foundation. Currently that stack - which is fabulous, it's a really neat piece of engineering; it looks like WCF, it behaves like WCF, so the programming experience is consistent and friendly and familiar – now currently it only implements the part of SSDL which looks like WSDL, but the framework is extensible enough so that you can then implement, which we believe to be the higher value aspects of SSDL, the pi-calculus base stuff that enables you to describe choreographies. Patrick has been kind enough to open source that and as of a couple of weeks ago there is now a SourceForge project where people can contribute and hopefully that toolkit will go on to become richer and richer and in my ultimate fantasy scenario it just becomes a de facto standard that people use when they are going to build WCF web services.

InfoQ: Any hope of a similar thing for Java yet?

Jim: It's something we have been thinking about (my colleagues at ThoughtWorks in Sydney, Josh Graham and those guys). What was meant to do it in fact when Indigo was being built we had the first skeleton of a project called Dingo a kind of tongue-in-cheek version of Indigo which we were

going to make SSDL centric and that has languished a little because we would have day jobs to do. My hope is that Soya, which is the WCF SSDL tool kits starts to get some momentum folks on the Java side, maybe folks like Arjen [Poutsma] who have seen SSDL and have spoken favorably about it may just build up the Java aside of the stack or the guys in the Ruby community may just build up something on the Ruby side of the stack. So it's optimism, maybe unfounded.

InfoQ: What do you see the industry moving towards with regards to these issues? Do you think that the RPC abstraction or the RPC style is going to still remain the most widespread style or do you think that the ideas are going to become more important?

Jim: I think as a developer, my day to day work is really constricted by the fact that the tools I'm given all reinforce the RPC mind set and I have to fight really quite hard to beat the RPC mindset down and try to make these current tools behave in a more messsagy way; over the years we've developed a bunch of patterns, for example, for using tools like Axis which are very RPC centric and being able to abstract away that there's RPC interface here and turn into a more message passing kind of system. I don't think most developers who are under the cosh will have the time necessarily or the inclination to do that and when the vendors come along and say: "Yes. Just take your EJBs, put them through this machine and out comes your WSDL and there is a SOA." I think they find that appealing because they have got a million other things they have to do, so I don't necessarily despair of the fact that we weren't ever moved toward a more asynchronous messaging environment. I think maybe we'll get burned a few times with some famous web services failures where RPC style implementation built out a system which is not very evolvable, which is high friction and so on, before people start to think: "Yes, I need better tooling." You see tentative efforts in this area, like the Spring Web Services stack for example. That's going to be a bit rude for a lot of developers but you can see the same kind of messagy ideas: "Here is a lump of XML, deal with it!" starting to percolate now into more mainstream frameworks so cautiously optimistic that it may not be a bleak RPC future.

InfoQ: When you mentioned that REST has this uniform interface with a set of operations and MEST similarly has one operation. Is this really an operation with application meaning? Isn't just an operation that is so uniform that it seizes to mean anything at all?

Jim: Absolutely. And that's the beauty of it. You forget that it's there because it's just a means of transferring a message from a sender to a recipient with the implicit request or hope that the recipient would process that message in some meaningful way in his context. So if I could get away having a uniform interface which has zero logical operations I would be happy with that. Unfortunately my own mental crutch was I thought services implement an operation "process this message for me please" and it takes "message" as its parameter and it returns "message" as a result. So for me that was kind of a crutch.

InfoQ: But in the REST world the idea behind that uniform interface seems to be that you can actually optimize the infrastructure based on those operations so you can do something different for a GET than you do for a POST. To do something similar on the MEST world you would have to look inside the XML and define somewhere what different types of XML messages you exchange.

Jim: Absolutely. In the REST world you are typically taking advantage of existing web infrastructure so

you can do idempotent GETs, you can cache the results from idempotent GETs and so on and get performance optimizations in that way. In a messaging world you can't, because we see the mechanism of transfer between two services to be a pipe, that may be a HTTP pipe, that commonly is today in which case maybe the transport does optimizations but we are specifically decoupling the notion of the message from the way it's transported and you can implement optimizations to the transport level but that doesn't affect the message payload and vice-versa so those issues are decoupled in the MEST world. Of course, I should use this opportunity to draw out an inconsistency within the REST camp: they assume a lot of REST practitioners are making use of the valuable features that the web provides, when the fact is that most people now are tunneling XML over HTTP or tunneling method plus parameters in a URL, and that's an even more horrible form of RPC than you can even achieve with SOAP and WSDL. In that case at least SOAP and WSDL RPC you can tool support to generate stubs and skeletons which is something you can't do with REST RPC. Trade market, just invented that. Mark Baker has been very good at advocating the benefits of REST, and yet now that is in the hands of developers that same kind of degenerative behavior that bugged web services, is now bugging the REST community. It's going to be an interesting learning curve for REST people to get on when they realize they have to start marking resources as cacheable to exploit the web; they have to have structured, readable URLs in order to identify resources in some sane meaningful way within their application context. And they can't just use HTTP as an XML tunnel because there are no benefits to that, over and above any other RPC technology..

InfoQ: When REST people criticize the WSDL folks, they point to WSDL 2.0, and say everything is better there and actually WSDL 2.0 can be used in some way to describe REST. Do you think WSDL 2.0 is an improvement over WSDL with regards to the RPC centricity?

Jim: Absolutely. I really think that WSDL 2.0 is better than WSDL 1.1. However I'm yet to see any services being built using WSDL 2.0. The cycle between WSDL 1.1 and WSDL 2.0 has been so many years. In fact I was working in the UK when I remember e-mailing the WSDL working group saying: "Let's call this 2.0." and that must have been at least three years ago and that is a long time to wait between releases; my concern for WSDL as much as I have concern for WSDL, which isn't very much, is that it has been such a long cycle that WSDL 2.0 has been in danger of being irrelevant or stillborn because WSDL 1.1 does the same things. Any matter of cleaner syntax, more strongly defined MEPs and so on really doesn't deal with what most developers are dealing with now, which is the operation abstraction which WSDL 1.1 covers reasonably well.

View Full Video：http://www.infoq.com/interviews/jim-webber-qcon-london

Related Contents：

- GET Details On Upcoming .Net Access Control Service
- SOA Transactions Using the Reservations Pattern
- SOA Meets Formal Methods
- REST Truer To The Web Than WS-*

Interview

Ian Robinson and Jim Webber on Web-based Integration

In this interview, recorded at QCon London 2009, Ian Robinson and Jim Webber talk to Stefan Tilkov about the Web as a platform for integration, the usefulness of various degrees of RESTful HTTP and the benefits of REST in theory and practice.

Ian Robinson is a Principal Consultant with ThoughtWorks, where he specializes in the design and delivery of service-oriented and distributed systems. Dr. Jim Webber is the Global Head of Architecture for ThoughtWorks where he works with clients on delivering dependable service-oriented systems. Ian and Jim are currently co-authoring a book on Web-friendly enterprise integration.

InfoQ: Welcome to this interview with Ian Robinson and Jim Webber, here at QCon London 2009. As usual, we'd like to start off with you very briefly introducing yourself. Jim, why don't you start?

JW: I'm Jim Webber, I work for ThoughtWorks in the UK and I'm currently writing along with my friend Ian here a really fantastic book on integration using the web.

IR: My name is Ian Robinson, I also work for ThoughtWorks, as a developer. I work with distributed and connected systems and I'm also writing a book with Jim. Hopefully it's the same one!

JW: Why don't you tell us more about that fantastic book?

InfoQ: If you talk about this book you mentioned, the title is Web Based Integration. Is this the same thing as REST or is it something different? Can you briefly explain what you mean by this? Give us the elevator pitch on that?

JW: Sure. REST is like the trademark of the high priestesses of the RESTafarian Kingdom and unfortunately, we're not ordained in that church. We are using "web" because it's a bit more encompassing, it's a whole bunch of techniques that we see used out there on the big wide Internet, some of which aren't necessarily as pleasant or lovely or performant or scalable or sensible as REST, but have utility outside of that particular architectural style. We're a bit more broadly trawling the web for interesting techniques.

IR: I don't think either of us are RESTauranteurs much as I think people some people think when Jim stands up and gets very angry, but we are broader in our approach. We're just looking for some very

pragmatic approaches to problems that we come across month in, month out with that kind of stuff we are doing.

InfoQ: I personally remember that a few years ago, when you mentioned REST, you got really strange looks and people wondered what you were talking about and this obviously has changed. At least people talk about it a lot these days and the tracks are visited very well and and the conference sessions are packed. What do you think is the current state of actual adoption in practice? Do people use it, actually?

IR: Yes, we use it. We use it quite a lot; or use web based approaches, some of which are more RESTful than others. It's not necessarily across the board in everything that we do, but we are introducing just very lightweight ways of working with the web with some of our clients and gradually, working our way out of that - that kind of RESTful stack, but beginning to split things up into resources and address them and then connect them and then actually start to drive applications by way of hypermedia. Yes, we are seeing an adoption in many different areas.

JW: I'd concur with that. I think in quite a reversal of fortune, if you like, from a few years ago, where REST would be laughed out of any serious austere enterprise, often now it's not at all comedic to go to a client and take a webby standpoint as you are your default. Indeed, my current clients, who are very interested in massively high performance systems would have potentially gone with traditional nterprise middleware if we haven't done some empirical experimentation and found that a simpler webby approach was actually just as well suited for their needs. Having done those empirical data points it sort of emboldens you a little - you kind of figured out that the web stuff works quite well once in one scenario, and then again in another scenario, and it embolds you to default to that when you are talking about distributed systems integration.

InfoQ: Are these different degrees - if you may call them that -, the different degrees of RESTfulness or of adoption of the REST things, is this also the way that you introduce REST or the web based integration approach into a company? Do you start with the first step and then gradually add more and more from that REST stuff?

JW: Sure. I though Lenard Richardson has a brilliant scale, a chart of RESTfulness in decibels and of ROYs or something - I don't know! Leonard partitions it from level 0, which is basically tunneling, through to level 3, which is the hypermedia stuff, and I think that internally is my mental model. I tend not to share that mental model with people, because they get fixated then on the kind of "REST inside" sticker that they want to apply. Instead, when I'm designing systems and building systems, I'm just trying to think about fitness for purpose and often, at the moment, I'm finding that fitness for purpose tends to fall on the lower end of Leonard's scale. They are kind of web-aware, but not necessarily hypermedia- centric services.

IR: I think Leonard's model is actually a useful way of talking to clients about REST because he starts off saying "Take any problem - the simplest way to solve it is to break it down into smaller chunks". What do we do? - We just identify lots of resources and give them addresses. So, he's not necessarily talking about RESTful things in the first instance, he is just talking about how to break up a problem. Then he is saying "If we do the same thing over and over again, let's just do it in an uniform way" - that's his second level, just use those uniform methods.

Then his third thing is "If we are doing something interesting or specialized, then do it in a specialized way" - that's where he starts to talk about hypermedia. You can actually talk to clients, talk to other people, just about breaking problems down into simple chunks, doing the same kind of things in the same way over and over again and then specializing any way necessary. I think that's a nice way of talking about it. Then, you can layer on some very particular things about REST or about using the web. It's a useful way of getting that conversation.

InfoQ: Do you think some of the things that the people consider an abuse of HTTP are also valid steps on that path? Is it just something that you have to do or is it avoidable? Do you have to at some time tunnel method invocations through GET to change something or is this just something you never can justify?

JW: The angry man inside me - often not too deeply buried inside me - would like to tear your head from your shoulders at this point and insist that these are terribily bad ideas. However, in the real world, we are finding techniques like tunneling verbs and URIs are used in certain hopefully bounded contexts, sometimes less bounded, which makes them dangerous. Certainly, in a bounded context, what enables me to get some rapid tactical solution to market quickly, I'm willing to accept them.

It's a kind of sticking cluster approach - I confess - and we always intend to go back and redress those, but I think having the kind of architectural - I call it architectgasm - and designing the world's most brilliant RESTful hypermedia cached super thing may be not the simplest thing that could work immediately. Maybe we could take those kind of ugly steps, like tunneling, to get us rolling today and then as our system volume expands, as its requirements become more sophisticated, as it encompasses more systems, as its reach grows, then we can think about migrating that to more RESTful partterns, which are demonstrably suitable for that kind of system.

IR: We really want to exploit a large installed infrastructure. Things succeed because the web is already out there, but we also begin to accept some of the constraints that are there, as well, some additional constraints that are just there in the way in which the web has grown. Those constraints exist now, like the browser tends to accept only two verbs, GET and POST, and very often, we'll end up building solutions that just have to adhere to those constraints, whether or not we're particularly fond of tunneling stuff.

JW: That applies to some of the intermediaries - the REST architectural style is preached as if the web is this perfect utopia, where everything understands the full extent of the HTTP uniform interface and pragmatically, that's not true. There are just some actors out there on the web, which don't understand some verbs, even though HTTP suggests that they really should. That's the limiting to out thinking about, for example the curve of cache maturity that Mark Nottingham is such a fan of telling us about.

Those constraints pose real challenges at us at web scale because the web doesn't behave the way that REST describes that it should behave, so we have to take some pragmatic shortcuts to make systems work. There is value in REST, but there is more value in having working systems.

InfoQ: I actually think that Roy Fielding would very much agree with you that HTTP and the current web is not a perfect REST implementation - it's something he keeps saying all the time.

JW: Good, because otherwise I'd get lynched by his posse.

IR: And yet, nonetheless, it's successful. We can work our way through that.

InfoQ: I just noticed that we have used the term REST without really explaining it, so there may still be some people who don't know what we're talking about. Can you just give us a 60 second intro to what is REST?

IR: It's "pick your path to adventure for interesting business processes on the web". We want to realize some goal having a couple of different things cooperating, we get to do that by serving up some HTML or some XML and the client or the consumer can begin, given a set of goals "I'm trying to achieve this thing or that", it can begin to pick its path through the server landscape picking up on links inside those representations and working its way towards the goal.

JW: That was really cute - I'm gonna steal that, we'll have to edit this to make it sound like it had been my idea. It's about the notion of servers leaving bread crums for clients to follow. It's leading the client through the business processes the servers implement. We tend to get bogged down in the kind of uniform interface and HTTP and all that stuff and really the heart of it is the server takes you by the hand and guides you gently through a business process.

InfoQ: You mentioned some stuff that's missing and all and that's problematic in the infrastructure, such as the browser limitation to GET andPOST and then the caches, intermediaries and other stuff ignoring or blocking some of those methods. Do you think there are other things that are missing currently in the current web space? Is there stuff that we should have to use this more effectively? If so, what would that be?

JW: Experience is the main ingredient which is missing here, although the web itself is a really mature technology. I think we are only now learning how to direct its particular characteristics towards integrated systems. The web has been brilliant as a mechanism for connecting humans, particularly in recent years, when humans have taken to the web in their milions to interact with pokes and tweets and all that kind of stuff.

As distributed system engineers we still lack that level of experience for doing the same things with computers. We haven't quite figured out yet in any robust way how to extend hypermedia, for example, between systems. For me, that would be the key thing that I'd say it's lacking. I'm happy to work around quirks in the infrastructure, differences of opinion around the community, but I think really we need to just experiment with this stuff, learn how to make it sing.

IR: Things even at the level of client library being able to surface hypermedia in a relatively common or standard way. I'm thinking I get a representation back, but I just want a link query that allows me to identify all of the hypermedia and then, based on whatever it is I am trying to achieve right now, I can choose to dereference in those URIs whatever pursue that hypermedia.

InfoQ: What are the good places to use REST and the web and what are the places where you should avoid using that stuff?

IR: Whenever you want real reach for your applications, then I think REST and the web are an

attractive proposition. It's a relatively low barrier to entry for anybody to be able to consume your application or work with it. Whereas if we are just working within Enterprise boundaries - whether it's a good idea or not - we are free to create our own idiom. If we are never going to have to explain that to anybody else, we could invent something from the ground up, but the moment we want to cross any of those organizational boundaries let's start looking for sufficiently sophisticated but nonetheless lowest common denominator way of working and cooperating.

JW: I can take that stage further and start looking at some of the architectural trade offs that present themselves when you're considering this use of technology and my favorite for the web is "Can you trade latency for scalability?" The web isn't a low latency system, but it's hugely scalable, particular the way you confederate load on the web. If you can afford latencies of seconds, minutes, probably about hours, days, weeks, the web is going to scale really well.

But if you can't afford high latency, then probably looking at a web inspired solution is the wrong thing and God will strike me for saying this, but some proprietary transport substrate with millisecond latencies or better may well be the thing you need. However, I've often found particularly techies will always insist that they need the millisecond transport substrate upfront without really holistically understanding the kind of business problem they're looking at, and the business problem may well call for something much more sensible, like seconds, in which case the web could be a sensible low ceremony way of achieving the same goal.

Geeks like us suffer terribly from the sin of pride because we always want the coolest, fastest, lowest latency, shiniest brass knobs-on on system and the web is really not about that. The web is like "hum-drum get on and do it". Trade latency for scalability any day of the week and if it comes out in terms of scalability with high latency, go with the web.

IR: I think there is a more general issue for distributed systems development as well. It asks us to think a little more about our tolerance for latency, for inconsistency. We've been accommodating these things for centuries. I can send a horse galloping of from one town to another with an order and some terrible things can happen in that intervening period. We've invented business protocols that can handle all of that and I think this kind of work is forcing us to look at those and to surface those protocols semantics again instead of always depending upon the low latency substrate and trying to delegate everything to the technology.

JW: That's interesting because the web as a distributed platform absolutely insists that we deal with distribution. From so many years in computing science now, we've been told abstraction is a great thing and we should abstract away all of that hard computing science stuff to the back room boffins and we should forget about it in living happy business web site land. Actually, you can't do that Waldo told us that years ago and he's been woefully ignored by the computing community, but when you decide to build a web based distributed system, the web doesn't hide that distribution from you. In fact, it gives you useful information to coordinate distributed interactions and, for example, to use the messenger-horse metaphor, to know when your horse is being robbed by a highway man at gunpoint and to take some corrective form of compensating activities. As a former transactions guy, I see the web as a big coordination platform - a kind of two phase consensus gone nuts.

IR: Just get over the fact you can't have a God's eye view of your success.

JW: Apart from apparently Sir Tim … he can see the whole web all the time. Seriously, you post a blog, he knows it - he is watching you. He is watching all of you right now through a webcam.

InfoQ: As you mentioned transactions, one of the critiques that I hear most often about REST is that there are so many enterprise features missing from it. In web services you have the transaction protocols: WS-Coordination with Atomic Transactions and WS-Business Activity and all that stuff. Do you perceive that as something that's lacking? Do we need a transactions protocol on HTTP that's RESTful?

JW: No, next question. I don't think so. I think we're learning the kind of scales that the web works at, the classic two phase transactions aren't really suitable. Anyone that listens to Werner Vogels talk about this eventual consistency stuff, anyone who's read some of Gregor Hohpe' stuff about how Starbucks doesn't use two-phase commit, anyone that has actually applied any fleeting thought about this understands that particularly two-phase transactions can't work on the web. You trade off consistency for scalability and the web's all about scalability, potentially eventual consistency.

If it's not too much of a blatant plug for the book, chapter 12 discusses this. Actually, we do bake off - at chapter 11 now we scotch the chapters so that we can keep up with Stefan's prolific pace of writing in his equivalent German book. We actually do bake off, if you like, in fact we use WS-* techniques for things like security, transactions, reliable messaging and so on. We show the equivalent patterns and strategies that we use in a plain old webby HTTP world. We don't claim that we're RESTful, we are just saying, for example transactions you don't really need because the web gives you all of this coordination all the time.

It's kind of perverse that the web being this synchronous step-wise textbased protocol it shouldn't really work at global scale, but it does because for each interaction I have with the resource on the web, I get some metadata telling me whether or not that interaction was successful. So, I can elect to follow the adventure route - if you like -, I can elect to keep going following resources and making forward progress or in the event of a piece of metadata that suggest that my processing is failing, I can perhaps take another route through a set of linked resources and other processes where I could make alternative progress. That, for me, is a much more sensible way of dealing with undesirable outcomes, trying to wrap everything in a big hawking transaction.

IR: "You are confronted by a dwarf with an axe. What do you want to do next?" I mean, even with the WS-* protocols, I don't think we should be tempted to use them all the time to try and coordinate and involve a number of different services in some kind of transactional context. It may be that you actually want to use that behind some coarse grained boundary and some internal implementation of service, even if we are exposing it across the web in a RESTful manner that the internal implementation might depend upon some of those lower level protocols. I think that's fine. If we are prepared to tolerate the expense of locking a number of resources. We are seeking a coarse grained boundary where we don't necessarily have to do that at that level.

JW: That doesn't come for free, right? That takes explicit clever design decisions to get right because at the lowest levels, if you are using one of these legacy relational databases, you are going to have to think about these things - yes, I said it and I stick to it, too! - but you are going to have to design explicitly and be very wary about your abstraction boundaries for those kind of details don't

inadvertently leak. If they leak to the web, you are screwed!

IR: Once you start giving somebody a key to your back door, they'll be in there.

InfoQ: Reporter: Now, that we've dealt with transactions, what do you think about having a BPM/BPEL-like thing for REST? Do we need something like that?

JW: Doesn't the web already have that built in its links? The web has all this choreography stuff for free.

InfoQ: Does it? I'm not sure. This is a real honest question. Those engines … never mind the program language used to program them, but those engines deal with things like coordinating multiple requests to multiple systems where the answers get delivered asynchronously and they coordinate them again and do something else. That seems like a useful capability. Shouldn't we have the same thing for the RESTful world?

JW: It is a useful capability. The notion of knowing an outcome that you want to get to and maybe some rules that will help you to get there is a fine thing. It's only when you tie it up in an inflammatory language like BPM, that it raises my hackles because that comes with a lot of baggage. We've all seen the kind of point and click ware BPM product are and we run screaming from them because they are dangerous things. The hardest point in using the web is the coordinating from the client side. If we could solve that problem, the web would be a much more amenable solution, but I completely agree that we need some kind of client-side coordination, but I don't think it should be of the same vein of the products and solutions we've seen today. Something like Prolog or a rules engine. might actually be a better way of dealing and orchestrating processes on the web.

IR: That can be an internal implementation issue for a client or for a server, whatever role they are playing at that point in time. It's not unreasonable to say in order to realize a goal, you might anticipate a few of the steps that you are going to have to go through. If your server is giving you back a representation, offers up a set of opportunities, you are applying some intelligence to that to pick your path, which does also suggest that there is an out of band mechanism as well, so that we can begin to communicate what is that you might expect to receive. It provides some reasonably standard interpretations of things such as "rel "attributes and stuff like that.

JW: That out of band intelligence could be a micro format. It probably should be because they are low ceremony and lovely.

IR: Yes, but many processes are very simple, sequential, or driven by events. It's relatively simple to implement them in the simplest fashion. It doesn't necessarily depend upon the rules engine or some work flow engine or anything like that.

InfoQ: Let's get to some practical things. We talked about theoretical advantages so let's talk practice. What kind of tools do you recommend to people who actually are convinced and want to build something RESTful? From the different technologies spaces that we have, what are your favorite tools to build HTTP RESTful systems?

JW: I confess I'm the fondest of very simple tools. I'm currently working on some rather high

performance systems and it happens to be in Java, which is fine. We have, of course, several choices in Java we could get with Restlet, we could get with JAX-RS servers implementation both of which are substantially sophisticated Frameworks that take out a lot of plumbing for us. In this case we went with servlets because they were sufficient for us to get the job done in a very low ceremony way. Flip side: if you are on the .NET platform, for example, you've got the WebInvoke and the WebGet stuff from WCF that you could use or you could just use a HTTP handler.

IR: Or a HTTP listener as well, which is actually what WCF uses under hood if you are self-hosting HTTP. You can drop down to that and again, it's very simple to build things on top of that.

JW: The rather slippery answer is that you take your pick. If you are comfortable with using a highly abstracted Framework like WCF or JAX-RS, if you contain that in your business domain more readily than you're prepared to tame something like servlets, which is very HTTP request/response-centric, then it's your call. Use what makes best sense to you!

IR: One of the things I'm often looking at is how I'm going to communicate something around the application protocol and typically, I want to communicate it by way of tests. Tests are a useful piece of documentation. By application protocol I'm saying I want to be able to describe to you how you can expect my service to behave if you submit this representation to this end point, invoke this method, then you might expect to get back this kind of representation, this media type, these status codes, this HTTP headers. All of those things form part of that application protocol - we are establishing some little contract between ourselves. You see a lot of this stuff in the AtomPub spec, for example.

What I'd like to be able to do is to assert all of that in a test. One of the things I'm often looking for is can I do that without always having to spin up an instance of my service or communicate with it over the wire, so I'm often looking for very very lightweight abstraction that allows me to create expectations against all of those HTTP artifacts, without actually having to start up an instance in the service. I know you've done it with some of the mock context in Spring. JW: With servlets and some of the Spring mocks it's actually a really nice way of not having to do the full bring up service wait 20 hours for Tomcat to come up kind of thing - very lightweight, very pragmatic.

IR: Whereas what I've done occasionally is create very thin wrappers around things such as a request or response. I can test independently that they actually do delegate to whatever runtime I'm using, but then I can basically write my tests against those or mock instances of those requests and responses.

InfoQ: You mentioned the word "contract". How do you see contracts relating to REST? Because in the web services world, the contract is really at the heart of everything. It's the great WSDL description that Jim is a very big fan of - as I know - that actually really describes very formally and very completely what methods your service exposes. How are you supposed to interact with a service that has no formal description? How could you possibly work with something without having that WSDL file?

JW: You have an informal description and then you have a bunch of Ian's fabulous consumer driven contracts.

IR: I'm thinking that very often the media type is expressing some kind of contract, is making some promises about the kind of representation you can expect to get back. The more interesting media types actually contain a lot of those more protocol-like rules as well. Again, I think at things like AtomPub that not only tell you what kind of stuff you are going to get back, but they tell you some of the methods that you can expect to be able to invoke and the status codes that you can expect to get back. There are contracts here, they are just being shifted around and I think we should be looking for media types that make very clear what is that we can expect to do, how we can expect to surface or interrogate these representations for hypermedia and how it connects us to hypermedia in order to progress an application.

InfoQ: Is it perhaps to say that hypermedia formats assume the role of contracts?

JW: Yes. In a nutshell, yes. In fact, a friend and former colleague of ours - George Malamidis - once said to me "The web already has a contract language - it's called HTML." I'm still scared when I say that sentence. George is a very sophisticated thinker in these circles, but I have a tendency to believe he is right. I'm just scared to make the leap to where he is.

InfoQ: Let's assume you have managed to convince some people that REST is a good thing, but they, in their turn, want to convince their co-workers to actually start it. Do you have recommendations? How do you go about evangelizing REST in your company? What's the best way to do that?

JW: I can't evangelize it. I think it has to be about a solution to a problem within a context. One of the systems I've been involved within the last year or so was originally penned to be based on JMS. That's great, I like JMS, it's a lovely idea, but the initial design was done without really any holistic thought to the environment in which the system was going to be deployed. JMS, lovely as it is, has its complexities. What we actually found was for the loads that we wanted to put through the system, by doing a small spike, few days worth of spiking, the HTTP was quite good enough for where we needed to be.

That had so many benefits in terms of improving our software delivery, it was a lot faster, easier to write HTTP things then it was JMS, they are easy to test with tools like Poster or curl, the delivery of that particular system was good and there is a man at the back of the room smiling about it, because he was involved with it and it was lovely and I feel that had we gone down the JMS route we would have to work so much harder to surface this system for testing particularly to our QAs. The fact that our QAs could bring in Firefox with the Poster plug in and probe the system, may be some really advanced but accessible exploratory testing and they broke us in wonderful ways that we hadn't expected because of the system surface area, which is open to them and that me smile a lot.

IR: It opens out to a larger constituency, doesn't it?

JW: Yes, so a reach thing again.

IR: Far more people having visible insight into the way in which the system is working or the way in which it exposes itself to the world. And they are seeing it in ways with which they are very familiar - they are looking at it in a browser, things like Poster and stuff like that. It's curious: we started all of

this saying that, in fact, we are more interested in talking about webby, Webbery things and the REST and then we continue to talk very much about REST, and I think to evangelize REST within an organization is occasionally not the appropriate thing to do. I always get frustrated when people say "We want SOA". SOA is another one of those words that should be under erasure. We should just start talking about what it is that we are trying to do and talk about it in familiar ways because very few people aren't now familiar with the web. We can just talk about some of the simple things that we do with the web and say "Imagine if your application could also work like this."

JW: There is the danger as what happened with SOA that it becomes bound up in products such to an extent where it becomes "I can sell you an SOA" - "No, you can't" and I think we are seeing already this REST moniker being applied to software products. It really confuses the discussion because people think they can just plug in REST, they can just buy REST platform and they are suddenly RESTful. Then all they are doing is tick the "REST inside" box and they haven't really given any critical thought to why that might be useful to their business. It's just the senior IT decision makers and the vendors conclude on a decision which is not necessarily in the business's best interest and it is rarely in the best interest of the development team who are trying to service that business.

IR: It's rare to be able to insert some kind of adapter and take a WS-* application and suddenly surface it as a RESTful application and expect it to be a rich and useful RESTful application.

JW: That's a dangerous REST application because the underlying implementation isn't designed to have such a surface area or to be loaded in that way, the design to be loaded in a message centric or RPC-ish way.

IR: I think there this huge in and of itself thinking of things in terms of resources and to try to layer resources on top of something that's been designed around an entire different paradigm. You are missing an opportunity to discover something interesting about your business, about your process. Discussing in terms of resources, often surfaces the value inherent in doing something. Search results in and off themselves are useful to companies like Google. It's one of the ways in which they monetize what it is that they are doing. Surfacing a search result as a resource is a good way of thinking and talking**.**

InfoQ: Audience Question: REST lies on top of HTTP, which is has quite old specifications. We've been using that for a few years and maybe would REST be cut back to what the HTTP specification was meant to be like. We are using the HTTP verbs in a more interesting way, despite the way we've been using that for the past 20 years, maybe, and still we have browsers or clients which do not implement HTTP specification fully. We know, for instance, it's very difficult to use Flex with REST - that's quite scary! What do you see in front of you? Do you see that we need a new specification, an update, so that we could also address problems that we didn't have when we were using HTTP as we have done, but maybe now we need also more power from HTTP? Or do you see that in 1-2 years all the browsers will implement the current 1.1 specification and we will be happy for the next 20 years?

JW: The primary reason why the human web doesn't support the full gamlet of HTTP verbs it's that HTML doesn't support it, so we are left with GET and POST support, which is a pretty limited

vocabulary. I'm not too worried by this because to me the browser are already dead. It's the most frequent, but the least interesting agent on the web. I'm much more interested in what happens when computers interact rather than when humans point browsers at web servers and right now, that infrastructure creaks at the seams when humans push it, but it's good enough for them to facebook each other or whatever it is that kids do nowadays, so I'm really not worried about it. What actually worries me more is some of the future directions that some of the working groups in the W3C are heading towards, which is effectively trying to rewire the web. Right now, the web infrastructure as it is, has got this magic tipping point where it is globally available, it has global reach.

I'm concerned if some folks at W3C come through and for example HTML 5.0 somehow makes it out into the wild, that we got this weird paradox - half the web is the original web and half the web is this new web and it's all got web sockets and it's all very confusing and it's not all mark up language any more and that's what troubles me most. Right now, I'm looking for the browser providers to innovate - I'm comfortable with that, I'm not passionate about it, but comfortable with it. I'm looking for the W3C to nurture the web in a more evolutionary manner and I'm not looking for someone to become Sir Tim the 2nd. Unfortunately, I'm concerned that some people in W3C are looking that way - hands off!

InfoQ: Audience question: Lately, we are seeing, even here in the conference, that in programming there was Lisp a long time ago and then we were going so much like we are trying to do more abstractions and we go to objects and big stuff components. Now we see that people are going back to functional programming. The same thing has been with the web: we got this simple HTTP specification, we started to build a lot of abstractions, SOAP and BPEL, and then we go back to simplicity, to REST. Is it like a trend now to go back to simplicity or does it happen all the time this way in software, to go back and forth?

JW: I'm not old enough to answer that question. Ian has seen several of these cycles, so he might have a proper answer.

IR: From the point of view of nostalgia-driven development, where every text begins well, wouldn't it be nice if we could do it the old way. As you were talking about simplicity and there being a drive towards simplicity, I think one of the benefits of REST evangelism - when it does take place - is not actually to insist on simplicity, but to insist on the constraints, to surface and recognize the constraints all over again. A lot of applications have been built on or around the web that abuse the web's infrastructure and the way in which it works. Good REST evangelism is surfacing and emphasizing some of those constraints and saying that if you work with or under those constraints, you will realize greater reach, better performance. That is a partial answer from me.

JW: You are right. We did put abstraction after abstraction onto our distributed system infrastructure and you know what: it hasn't worked out that well for us. Some of the largest and most sophisticated distributed systems on the planet haven't been all that large or sophisticated and then this kind of crappy protocol comes along that insists on being synchronous, and insists on being text-driven and it scales globally. That's shocking and does not make sense to us as engineers. That's the web paradox - it's the rubbishest thing on the planet, but it's scaled and for me that is what's hit the reset

button because I was totally up for XML-based protocols that do all sorts of funky stuff.

I put my name to some OASIS work and some other stuff in the transactions phase - God forbid! -, but to be fair, we thought we had the best of intentions, we thought this stuff was going to be useful and it may still be useful in certain bounded context, but what the web and HTTP have shown us is that if you want to scale and reach out globally, you have to have something that's dumb. Dumb protocols are the base line through which everyone can interact and getting that interaction seems to be now what's critical in early 21st century computing. So - Yes, back to basics.

View Full Video：

http://www.infoq.com/interviews/robinson-webber-rest

Related Contents：

- REST is a style -- WOA is the architecture
- HATEOAS as an engine for domain specific protocol-description
- How Relevant Are The Fallacies Of Distributed Computing Today?
- Presentation: Transforming Software Architecture with Web as Platform
- Presentation: REST: A Pragmatic Introduction to the Web's Architecture

Interview

Mark Little on Transactions, Web Services and REST

In this interview, recorded at QCon London 2008, Red Hat Director of Standards and Technical Development Manager for the SOA platform Mark Little talks about extended transaction models, the history of transaction standardization, their role for web services and loosely coupled systems, and the possibility of an end to the Web services vs. REST debate.

Dr Mark Little is Technical Development Manager for the JBoss SOA Platform, Red Hat's Director of Standards and representative on the Java Executive Committee. He has over 20 years of experience working in the area of reliable distributed systems. While at Red Hat/JBoss Mark has been the lead of the JBoss ESB and JBoss Transactions products as well as working from the office of the CTO.

InfoQ: This is Stefan Tilkov at QCon 2008, and I am interviewing Mark Little. Welcome Mark! Can you tell us a little bit about yourself and what you do?

Mark: I am a Technical Development Manager for Red Hat's SOA platform, which basically means I am involved in all our SOA strategy. I have various groups reporting to me like workflow, transactions, ESB. I am also a Director of Standards, so I am also responsible for participation in W3C groups, OASIS and JCP.

InfoQ: Ok, so if I read your name somewhere the one thing that pops up in my mind, the one thing I associate you with is transactions. I think you have a long history in being involved with standards around transactions. Can you give us a little background; can you actually define the term transaction for us, give us a little refresher there?

Mark: So "Transaction" is a term that is misused probably a hundred and one different ways in our industry. It's probably better to be a bit more explicit and say it's atomic transactions. An atomic transaction has its history back in the '60s, and it's hard, it's like it's a fault tolerance mechanism. It's based on work that was done around that time in the '60s on spheres of control. Basically an atomic transaction is a sphere of control, it's an activity that guarantees that work done within the scope of that transaction is either all done, or it's not done at all, you get no partial failures.

A typical example would be a bank account system. Suppose you are transferring money from a current account to a high interest account, and one way of doing that without using transactions would be to remove money from the current account, so you are holding the money and then deposit it into the savings account. If you have a crash of the system at doing this, then depending on

where the crash happens, you may lose your money. If you take the money out of the current account but it hasn't quite been put into the high interest account yet, where is it? Obviously the bank hopefully would have information about where it is, and they can do some kind of money resolution, but that could take days to actually sort out and meanwhile you haven't got your money, you can't pay your bills. If you are to do that transfer within an atomic transaction, then the atomic transaction would guarantee that if there was a crash the money goes back into the current account, the transaction system would do that, or depending on where the crash is, it would guarantee that it would eventually turn up in the high interest account, hopefully within a matter of seconds or minutes at most. And it would do that itself automatically, there will be no requirement for manual intervention.

InfoQ: One of the topics that come up often in discussions about web services is whether or not they need transactions. Maybe we can start by a quick description of what is actually available for web services in the transactions space.

Mark: Web services transactions development has been going on for almost as long as web services has been developed. So I started doing work around that in 1999, which is pretty close after SOAP was first released. What we were looking at then and what we have continued to look at over the intervening eight or nine years is actually a number of different approaches for transactions in the web. So, traditional atomic transactions that I described earlier have some in-built assumptions about how they will work and the environment in which they will work, so pretty much they assume they will work in a closely coupled environment that can be trusted and they last for seconds, milliseconds hopefully, but seconds, maybe at the utmost minutes. On the web those kinds of interactions typically don't happen, you know you might be booking a night out, or buying a book from Amazon and you might be doing that over the course of hours or days. And to do all of that within the scope at the top level of an atomic transaction, just doesn't work. So we were starting with how do we do transactions that are specific for web services or for these long duration interactions. And there was a lot of work that was done back in the '80s and early '90s on what is known as extended transactions.

There is a range of extended transactions. Basically the principle about extended transactions is to relax the very properties that are inherent within an atomic transaction, so if you go and look at the literature then you'll find that another acronym that is put around atomic transactions is also known as ACID transactions. That is ACID - A for atomic, everything happens or nothing happens, C for consistent, the state of the system moves from one consistent state to another, I for isolation, so you can't see dirty data and D for durable, so that if the work happens it is made persistent even if there is a crash, you'll eventually get the same state. Extended transactions relax those properties, so you might relax atomicity, so when an extended transaction or a certain type of extended transaction terminates, you may say "I want to commit but I don't want to commit two out of three of these participants, I want to commit that one, but the other two I actually want to undo". Another extended transaction model might relax isolation. And the reason for relaxing the different properties is to cater for the type of use cases that you want, and that's why there is a lot of different extended transactions models. There is no one model that actually fits everything you could ever want to do.

So that's what we have been doing over the last eight years, we have been looking at extended transaction work that has been done and trying to come up with a way of allowing people to develop extended transaction models that are good for their particular use case, rather than try as a transaction industry has done twenty years prior to this, shoehorn the ACID transaction into absolutely everything, let's have targeted models, targeted implementations, and we have got there. So it has taken eight or nine years to get there but finally in OASIS there's the WSTX technical committee, which has defined a framework, WS-Coordination, which allows you to plug in different intelligences, so this would be the different types of extended transaction models.

Out of the box, the standard provides two extended transaction models, because of the use cases that we currently have that we need to adopt. One is Business Activity, which is for these long running units of work, the other is Atomic Transaction, so despite what I said earlier about atomic transactions not being good for web services, if you can recall that back when web services where first starting and even through to today, people are using them for interoperability, as much, if not actually more than for Internet scale computing.

So, atomic transaction in the WSTX spec is really there for interoperability between heterogonous systems running on closely coupled networks. You could use it across the Internet, there's absolutely nothing to prevent you from doing that, but there are really good reasons why you shouldn't.

The Atomic Transactions spec in WSTX has given us transaction industry interoperability between obviously Red Hat, IBM, Microsoft, and a couple of other companies. All heterogeneous transaction protocols within about a year and a half of the spec's being finalized, probably less actually, whereas if you are looking when we last tried to do this, which was in the OMG within the Object Transaction Service work, that really took us about ten years. So there were definitely benefits for doing it in web services.

InfoQ: So you have some practice now, so maybe it's no wonder it took you less than ten years to start.

Mark: Yes, you are right, we did learn from our previous mistakes.

InfoQ: So you said that on the Internet you would never use atomic transaction, which is pretty obvious, but I believe some people would claim that even if you have the ideals of building a service oriented architecture then loose coupling becomes a design principle even if you stay within the company's boundaries. Would you also say that if loose coupling is one of your goals, atomic transactions is not a good match for that?

Mark: Yes, I would, but some people still want to do it. So you can make recommendations but ultimately if they want to do it then nothing in WS-Atomic Transaction will prevent them.

InfoQ: Many people say transactions and SOA, transactions and loose coupling don't mix at all. What are the benefits, how would you actually advocate the usage of those pretty complicated standards to people who use them?

Mark: I think some of this comes back to what I said at the start about the use of the word "transaction", a lot of people when they see transaction they immediately assume ACID transactions,

two phase commit, database transactions, however they've run into transactions before. And in that case they are right, I would not recommend to customers to use atomic transactions across the Internet or within the corporate firewall if what they are trying to achieve is a service oriented architecture based system.

But if you look at extended transactions, like I said about the relaxation of different properties, you are going to actually see that there are certain ones that are actually good for SOA based applications, they provide you the guarantees that you might want but they don't provide the restrictions that ACID transactions require. So if you actually look at some of the work that we did in WS-CAF, the web services composite application framework, which predated WS-TX, there's one of the transaction models there, that unfortunately we didn't adopt into WS-TX, which is actually much more relevant to SOA based implementations.

It's the WS Business Process Model, and I think we actually started that back in 2003, and since then companies like Microsoft and obviously Red Hat, and other companies, are talking about very similar things where you no longer have this notion of global consistency, there's no notion of "everybody has the same state", because in a large scale system you can't guarantee that. Well, you can guarantee it, but it might take you until the heat death of the universe to make sure that it is the case. And that is exactly what WS-BP did, assumed that there were these domains of consistency and in between them there might be domains of inconsistency, up to a certain level or even fuzzier than that. So for people looking to use transactions in a SOA based environment I would suggest that they don't come into it with the preconceived notion that transaction equals ACID transactions or atomic transaction. "Transaction" is too over-used, there are extended transaction models out there that can be of benefit to your application.

InfoQ: One of the things you have briefly touched upon is that it is actually WS-Coordination, and WS-Atomic Transaction and WS-Business Activity, which are essentially the Microsoft-driven standards, that have been incorporated. Is that a correct view of this? Because I actually remember that there was a sort of as usual in the web services, or what used to be in the web services space there was a wall between two different fractions, maybe you can give us some background on that?

Mark: So, the work actually on extended transaction being standardized, started back in 1997, in the OMG with Arjuna, the company that was with us at that point, and IBM and a few other companies working on something called the Additional Structuring Mechanisms for the OTS, rolls of the tongue, which is short hand to the CORBA Activity Service. That was developing a framework essentially a pluggable coordinator, where you could add the intelligence for your specific transaction models. If you map that to what we have in WSTX you'll see that there is almost a one to one: the pluggable framework and the OMG spec is essentially WS-Coordination. And the intelligences were the different protocols. So for WS-Transactions there is an equivalent mapping in the OMG spec.

When we actually started to do work on the web services transactions back in the 1999 we were working with IBM on essentially taking this model and adapting it to web services. But you are right, at that point the web services wars between Microsoft and IBM fighting Sun, Oracle and pretty much everybody else, they kicked off, and IBM and Microsoft went their own way, building on this work,

and we went our own way again building on the same kind of framework, and what came out of it was WS-T from IBM and Microsoft in 2001, and WS-CAF which came out in 2002. Eventually everybody kind of kissed and made up, and we had the OASIS WSTX TC that formed, but the principle input to that was still WS-T and WS-CAF, despite the fact that I am a co-author on both of those original specs and obviously the standards as well, WS-CAF was still better I think. It was much more SOA based and much less "CORBA with angle brackets", if you like. And it's died a death now, but yes the TX one is the one that we are stuck with.

InfoQ: Sounds like a VHS/BetaMax story.

Mark: It is and the BetaMax was the better one. Don't even mention Blue Ray!

InfoQ: Given that web services support - for some level of support - transactions, would you see this as one of the benefits over REST? You know there is one continuous topic that comes up.

Mark: So we did add transactions to REST when I was working at HP in 2000, we actually did some work on trying to standardize a transaction-based, REST-based protocol. And we did it and looking back I am not sure if it was a hundred percent REST, I actually think it's probably ninety five percent REST, but anyway. The reason we did that is because we actually had customers who were coming to us and say "This web services stuff is a little bit too new for us at the moment. We are not too sure"; it was Axis 1.0 days, Axis didn't perform very well, I'm not even sure if it does these days.

"We would like to do something about coordinating multiple updates to web servers, we are using HTTP, sorry multiple web servers, we are using HTTP, can you do something for us?" And we did it, we did it for these customers, but it never got progressed and when I left HP it pretty much folded, went on the shelf, because web services were big or at least were getting big, and it hasn't really been any push at transactions back into vanilla web if you want, or REST or HTTP. I think that's because people are diverted towards web services; it's not because I don't think the requirement is no longer there, I have come across a few companies over the last five or six years who have asked similar questions, but they've gone eventually either for not using transactions at all and chancing to luck, or they have gone with web services. So I think the need is there, I don't think it's a huge need, but then there isn't a huge need for transactions anywhere.

I think that it would be nice if there was a standard, I think this is one of the problems with REST over HTTP at the moment, in that although it's a standard there is no generally agreed upon standardization of protocols that might sit on top, like transactions, like group communications, that sort of things. And hopefully if we can put these REST and web services wars behind us and kiss and make up then maybe we can actually take some of the benefits of both systems and standardize things in REST over HTTP. That's some customers are actually crying out for.

InfoQ: What is your opinion about the REST vs. web services war? While we are at it we might as well address it? What is your opinion on that?

Mark: I think it's going way too long and I think it's become very polarized in some sectors when it shouldn't have. There are certainly good reasons for using REST over HTTP, so obviously there is a distinction between REST and what I would like to call REST over HTTP, which is one way of doing

REST.

There are also good reasons why you might want to use web services. And I think for web services it really is the interoperability and the fact that everybody has got together and we have standardize these high level application protocols that sit on top. I don't think it's an either/or situation, it might be in certain cases, it might be that it really does make sense to use REST everywhere in a particular deployment, but I think in general and if you look back over like forty-odd years of distributive systems development, there has never been a global panacea for distributed systems. One thing does not work well for absolutely all distributed system or component within distributed system like you might come up with.

RPC has worked very, very well since the'70s, and we are revisiting that over the last ten years or so, but RPC isn't dead, people are saying RPC is dead, it is not, if you actually look around RPC is running a lot of backend infrastructural systems that are always going to be RPC-based. So when people come and say "You need to change this to message oriented" that is not going to buy those companies anything, they are happy with what they have got. And I think the REST and web services wars should pretty much just end; let's agree that there are good things and bad things about both, and let's try and use them together if we can. I think they can actually be used well together, the work to merge them together and make them work efficiently together hasn't been done, I think it could be done, and let's just get on with it, let's just stop too much fighting.

InfoQ: Could you become a little more specific, so what is good in web services that should be adopted in the RESTful word and what is good in the RESTful world that should be adopted in the web services world? Is there such a list?

Mark: Transactions.

InfoQ: Which nobody needs.

Mark: Yeah, there isn't a huge need for transactions, but there is a need for transactions. Security, high availability, WSRX for instance, you can build on that to do high availability services. From a REST perspective, the uniform interface does make a lot of sense in many cases. I think one of the problems that we have with web services is WSDL, to be perfectly honest. Certainly when I started doing work around web services back in 1999/2000 WSDL was still in its infancy and when we were developing specs and actually doing implementation in HP, we were developing on SOAP over HTTP and back porting the WSDL afterwards because the WSDL really got in the way, and I think it still does today. So getting away from WSDL and looking at the benefits that a uniform interface can provide rather than a specific interface, and also try not to abuse transports, so saying that SOAP over HTTP is the same as SOAP over TCP/IP because HTTP is most obviously a transport, it's most obviously not a transport.

InfoQ: You mentioned that it is now time to go beyond the wars between different facts. I mean that's probably something that has been going on for years in different areas, it's been COM vs. CORBA, and it's been these kind of web services versus that kind of web services and now it's REST versus web services. Given the last debate what would be your suggestion, what do you think should be worked upon, what are the things that we should address to get those two to unite, kiss

and make up?

Mark: I think that the web services guys, and I kind of include myself in this group, need to realize that certainly interactions across the Internet that are based on HTTP are more likely to be REST based than not. And bridging between web services and HTTP or REST over HTTP I should say, should be addressed in a more efficient manner. We shouldn't try to bastardize HTTP anymore as a transport, we should try and work with it rather than against it, and I don't think we are really doing that at the moment. Web services uses HTTP for a really good reason, and it's so that you can tunnel through firewalls.

And that's the real reason. I was at the first OMG meeting where SOAP was brought to life, and it was one of the debates about why it was there. I think we kind of progressed that we are doing it for the same reasons over the last seven or eight years. But I think that as a web services community we can do more to embrace REST than we are currently doing. I think we should. I don't think there is anything technically that would prevent us from doing that. If you actually look at the way the web works, there is nothing that should prevent us from being able to use true web protocols with nothing else laid on top of them, to actually talk between web servers across different continents.

Like I said before about the extended transactions stuff that's the WS-BP spec that I mentioned which has this notion of loosely coupled domains of consistency with inconsistencies between them, that's kind of very similar in that what happens within the corporate firewall might well be a combination of CORBA, REST, web services, DCOM, Java RMI ... whatever you want. But between the corporate firewalls, I think it should more likely try to be not REST we should actually work with that rather than try and fight it.

InfoQ: Would you say that a lot of the arguments that were dismissed within sixty seconds five or six years ago that the REST people made are now being accepted pretty much by everybody in the web services world? I mean I personally see lots of people like you, people who write the standards, who are involved at least conceding that REST is a good solution for many cases? I distinctly remember five years ago when it was said that it was only usable for browsers, for human to web server interaction, no machine to machine could ever possibly work within HTTP. That seems to have changed?

Mark: It has changed.

InfoQ: Could it be that they were just right?

Mark: To speak personally I did transactions over REST back in 2000, it's not like I certainly had a epiphany moment six months ago. I think a few other people have been kind of coming around to this, and yes whether you want to say that they suddenly realized that what other people were saying was right or whether they always knew it was right, and they just had different masters at that time who wouldn't let them say what was really going through their mind, I obviously can't speak for everybody.

InfoQ: One of the basic principles of web services is this protocol independence, is the independence of particular transport protocol, which is one of the big elements in the sales pitch

for web services. Doesn't that make the option of consolidating HTTP the way it was supposed to be used impossible. Is there really a way to consolidate web services in Restful HTTP?

Mark: I think there is. I am not necessarily sure that it's SOAP over REST. I think that again going back to what we were doing with transactions on REST back in HP, we were actually working to bridge web services transactions to RESTful transactions. And you could do it. I like to think that if we been allowed to finish it maybe it would have been one of those little gems that would have grown and maybe unified people around that fact that web services and REST can be used together. I do believe that we can do it, I'm not suggesting that it is easy, but if you look back at the amount of time and effort that has been wasted in these fights that we've had from individuals to big corporations, I would like to think that if we'd actually spent that time actually talking and trying to get these things resolved in a reasonable manner we could have been there by now.

InfoQ: There are lots of rumors going on at the moment about those two companies that happen to have merged recently, which happen to be JBoss and Red Hat. Can you give us a little bit of background on that? Tell us a little bit about whether everything is just nice and all those rumors are just crazy little things that we shouldn't believe? Are you willing to talk about that?

Mark: Red Hat acquired JBoss in July 2006. It definitely wasn't a smooth transition but I certainly didn't expect a smooth transition. But that wasn't because it was JBoss and it was Red Hat, I have been involved in more acquisitions in my career that I care to remember, and none of them have been particularly smooth. So it didn't come as a big surprise to me, I think one of the big problems though which is probably specific to the JBoss/Red Hat acquisition was the culture within JBoss because of its history of having to fight against the man, who was IBM or who was Oracle, it was much more combative than Red Hat. Being assimilated into a company like Red Hat, that had a different kind of culture like that, did cause friction.

I won't go into specifics but things like being a very private company as well, various mailing lists that people would talk within JBoss to each other throughout the whole company and use very colorful language, for a start you probably wouldn't want to do that in a public company anyway, but also it can offend ... the larger the size of the company the more chance it is it would offend somebody. There were those kind of things you have to be careful about what you say, your level of freedom has gone down a bit as a result, but obviously it's a bigger company, it has more money, has a bigger reach, so there are tradeoffs.

I think overall it has been a good thing, I think. Some people have left for one reason or another, sometimes because they didn't like the culture change, others because they wanted to stay with the startup mentality and not go to a big company, so they gone to other startups. There are still the odd culture clash within Red Hat and I think quite a few people, old time Red Hat people see JBoss guys as upstarts and brash and trouble makers, and treat them a bit like that as well and that doesn't go down well with some individuals who are brash and upstarts. They push back. There is a lot of to and throw but I think the analogy is probably teenage son versus forty years old father, JBoss is going through puberty and the father is the one who has been there and done it before.

View Full Video：

http://www.infoq.com/interviews/mark-little-qcon08

Related Contents：

- OASIS Releases a Raft of New Standards
- Web Services Test Forum Announced
- InfoQ Minibook: Composite Software Construction
- WS-TX 1.1 standard announcement
- WS-TX as an OASIS standard

Interview

CORBA Guru Steve Vinoski on REST, Web Services, and Erlang

In this interview, recorded at QCon San Francisco 2007, CORBA Guru Steve Vinoski talks to Stefan Tilkov about his appreciation for REST, occasions when he would still use CORBA and the role of description languages for distributed systems. Other topics covered include the benefits of knowing many programming languages, and the usefulness of of Erlang to build distributed systems.

Steve Vinoski is a member of technical staff at Verivue, a startup in Westford, MA, USA. Recognized as one of the world's leading experts on CORBA, he was previously chief architect and Fellow at IONA Technologies for a decade, and prior to that held various software and hardware engineering positions at Hewlett-Packard, Apollo Computer, and Texas Instruments.

InfoQ: I'm here with Steve Vinoski, one of my childhood heroes. What are you up to these days?

Steve: I can't really say what my company does. I left IONA Technologies in February and the new company is in stealth mode, so the founders don't want any details about it to be leaked out, but I can tell you that I'm having a lot of fun. It's like a breath of fresh air. This is very different compared to ten years at IONA and I'm having a lot of fun.

InfoQ: Can you tell us if it's in any way related to middleware or some new kind of Distributed Objects?

Steve: No, it's a totally different industry. I started life as a hardware engineer so there are some hardware guys involved and it is sort of back to some of my roots. I'm not working on the hardware, but there is middleware work involved. I've gone from being a vendor to being a user.

InfoQ: One could say that maybe this is reflected in the statements on your blog, which is fortunately available again. You said some not too nice things about vendors, middleware, WS-* and ESBs. Can you elaborate a bit?

Steve: I think if you go back and read my columns from Internet Computing back four years ago (in fact the first REST column I wrote was five years ago) some of them have been like this: "This is a good way of doing things using WSDL as an abstraction". Some of the other columns said: "This is not really standardized; there are too many specs, and all the usual vendor wars and clashes". I haven't really been kind to it all along but I couldn't really say what I really felt being part of IONA because that was their business. I think once that weight was lifted from me I became able to say what I really felt. It's not too far off to what I said before it's just that now it is completely honest, I have no agenda.

InfoQ: If you were now an architect in a large company faced with designing an architecture for a set of systems or a large distributed system, what would you chose?

Steve: I would look at REST to begin with. If you look at SOA it is more about business, about culture. It's all about how do we get our business to work together, how do we make things work together and make shared components that we can all reuse and how can we avoid duplicating effort and stuff like that. It's more about culture than it is about technical architecture. Some people talk about technical SOA, but technical SOA really depends on the product that you're using because every product is different. SOA isn't specific enough from a technical perspective to make them all look the same. Then you turn around and look at REST and it is a whole new architectural style; it's all about constraints and what you get from applying those constraints. Someone has gone to all the effort of applying a bunch of constraints to a distributed system and getting the desirable properties as a result of doing that. Why should I go and think I can do any better? The work has been done for me, and it's also defined loosely enough that if I have to tweak those constraints, I can do that. Just from a pure engineering cost perspective it makes sense to look at REST in my opinion.

InfoQ: So what would be the use cases where you'd use CORBA?

Steve: I would use CORBA if I had to talk to something already written and using CORBA. I started working with CORBA in 1991 and it is still around and I just got a royalty check from the book that Michi Henning and I wrote recently. Is not as much as it used to be but I'm not going to turn it down. There are industries that still use CORBA and those interfaces are not going to go away tomorrow, they are going to be around for probably 5 or 10 years. If I had to talk to something that was built using CORBA I'd use CORBA. If I was doing some very small scale system that the developers were familiar with the approach, I would use it, but if I had to build an enterprise scale system I would look at REST.

InfoQ: If we are to talk about one difference, one topic that comes up often in the discussions about REST is that there is no description, no contract apart from the one defined in the REST dissertation, which is the generic one. Don't you perceive this as a problem because CORBA is so strong in this regard with IDL? Is this something that's missing?

Steve: I've had a lot of thought about that as you might imagine. In CORBA there's obviously different layers, different areas that one can work on. I've worked on pretty much everything but when I was working on CORBA I've focused mostly on IDL and mapping it to languages. For what it was, I think we did a reasonable job. I know there are a lot of people who have a problem with C++ mapping, but it is written for very strong C++ programmers. I personally don't have any problems with it. There's a problem if you have to define something in IDL just to know how to use it. That doesn't really work. No one takes an IDL and says: "Here's this method, I call this and I pass this. I just look at the IDL and I know what to do".

Nobody does that. IDL is really for code generation. If I want to know how to use a service whether it has IDL or not, I go talk and talk to the developers, if they're nearby; if they're not I look at their documentation. So if you think about the REST services of Amazon or Google or any other site. They have documentation on the web, I go look on the web, I read it and I figure it out. I don't know if having an IDL would help. The interface is fixed - it's HTTP verbs. You have to deal with data

definitions and the data definitions, the media types are usually defined by registered IANA types; if you want to know how the data looks you go and look at those media types or MIME types. I don't see it as being the same kind of problem as the CORBA style of Distributed Objects.

InfoQ: One of the main arguments I hear is that if you use a typical statically typed language like Java or C++ then from the code generation step what you get is type safety when you build up those objects that you exchange when you call those implementations. If you don't have a description language that can generate the code, you don't have your code completion in your IDE and all the stuff that we've gotten used to.

Steve: I supposed there is something to that, but I don't use IDEs. I've been having a discussion with a former colleague of mine about that in my blog comments where they said "You should be using IDEs and everything". I've always used Emacs. I've tried using Eclipse and it does some things nicely but I guess I'm just an "old dog". When it comes to the type safety problem you can call it pseudo-type safety at best, because I can take a message that was supposedly type-safe in my client application and send it to your server and your server can be compiled with completely different definitions and still be able to read those bits off the wire and somehow they look like they fit your message definition, where the two definitions could be completely different. Similarly your object or service or whatever it is that I'm getting type safety from using an IDL could have completely different type in reality than what I have in my client, because it is all distributed. Your versions change at a different rate than what mine change at ... it's sort of pseudo-type safety at best. But I think that whole thing turns the whole equation because you're building a distributed system, you're not building a local program and distributing it, but you're building a distributed system and you happen to be writing pieces of it with the language that you've chosen. I think the focus should be on the distributed system and making a particular language easier to use in that context is the wrong focus. I know a lot of people disagree with me.

InfoQ: You spent quite some time discussing dynamic languages. Can you elaborate a little bit on that? I wouldn't have expected it from an old C++ programmer to suddenly switch to Ruby.

Steve: By "old" you mean that I've been using it since 1988, right? Not that I'm old ... I've been a C++ programmer for a long time, but I've also been a dynamic language fan for a long time. My degree is in Electrical Engineering but I've never taken any form of computer science classes. I always felt I had to learn computer science on my own and back when I was teaching myself different languages, C & C++ primarily, I didn't have anyone else around to bounce ideas off because I was in a hardware group. When I joined Apollo Computer in 1987 I started working with some software people, but they were primarily embedded developers mostly using Assembly language and some using C. I started using C++ and that just freaked them out. C was radical to be used in that environment, and C++ was completely off the charts.

I didn't have anyone to bounce these ideas off. Maybe I was missing something, I should be looking at all kinds of languages, not just these. So I just studied languages constantly on my own. I looked at pretty much everything. Not that I developed real applications in them, but at least I read books about it. I also got involved with Unix early on. There was a hardware test machine that I had to use; I had Berkeley Unix running on it so I learnt Unix on my own. Learning all the tools of Unix, the greps

and the seds and the awks, and when Larry Wall came out with Perl I looked at it and I said "Well there's this all this other stuff I've learned but it's all in one language". In 1988 I ported Perl to Domain OS, which is its Apollo's operating system and I think if you still find my name in the Perl source for doing that. The dynamic language stuff goes way back to the same year I started using C++. It's not a new thing; I've done it all along.

InfoQ: When you mentioned that instead of using CORBA you would now use REST, is the same true for the language thing as well? Would you now rather use Ruby or another dynamic language instead of C++ or Java?

Steve: I do tend to look at those languages first; sometimes they are not the right language. What I like to do is take multiple languages and just have them at my fingertips and look at a problem and say: "What's the easiest way to solve this? What language would make this easiest to solve?" Not only to solve but easiest to maintain going forward, easiest to extend. I look at the problem domain, I look at the languages you have in your tool box and chose the right one. While I prefer dynamic languages just because they are so capable, they are very brief; you can write programs that are at least an order of magnitude smaller than Java, C++ or C and still do the same thing. They are fast. People tend to say they are slow but that's not usually true. Some are slower, some aren't. Python is very fast. I don't rule our Java or C++. I'm not a big Java fan, to be honest, because if I want to use something like that I think I will go to C++. If I want something that's totally different than C++ I go to the dynamic language side. Java for me is too close to C++ to make that much of a difference.

InfoQ: You spent a lot of time playing with Erlang recently. I don't know whether playing is the right word, but I saw you implementing Tim Bray's Wide Finder. Can you give us a little background both on the Wide Finder idea in general and on your experience with Erlang?

Steve: I've been looking at Erlang for a couple of years actually. I haven't been using it for a couple of years, but probably two years ago I started seeing references to it. Usually someone says "There's this language you should look at it" and my initial reaction is "Ok I will take a look". If I don't see an immediate use for it, I'll get back to my real work. That is what happened, but it sort of intrigued me because of the reliability and concurrency aspects that it has. Being a long time middleware developer I spent a lot of time trying to make sure that things are production-hardened. Getting messages from here to there, translating data, that's the easy part.

It's when the thing has to stay up, it has to fail over in case of problems with one of the nodes, or you need fault tolerance, all the reliability issues, and then the whole concurrency thing which is where you spend a lot of time just figuring out ... I've got a lock this piece of data, shared across these threads and if I miss one bad things are going to happen. Those are two hard problems areas middleware developers deal with constantly. I look at Erlang and it is sort of built in. That may bear more investigation so I sort of kept looking at it. When I was at IONA I was working on the advanced message queuing protocol implementation that Apache is working on; it's called the Qpid project. I was working on that and someone asked me to look at making it fault tolerant. I said "If you are going to make it fault tolerant you should be doing it in Erlang, it would save a lot of trouble."

Two weeks later a company called Rabbit MQ comes up with an Erlang version of AMQP. They had obviously been working on it for a while. It's still around and people are using it. I guess I wasn't too

far of the mark there. When it came to Tim's Wide Finder ... Tim Bray works for Sun and he wanted to analyze his weblog; probably a quarter gigabyte of data for the smallest log, a lot of data to analyze. He thought at "Sun has this new machine coming out. How could I make use of a language like Erlang to parallelize the analysis of this data?" He wrote an Erlang program and he was very unhappy with it. If you go back in his blog you can see he's quite unhappy and he thinks Erlang is not what it's cracked up to be.

I saw that and I thought that maybe I can do a little bit better. I started working on it, other people in the Erlang community were working on it. We just saw the time dropping. I think Tim's initial stab was at 30-40 seconds to analyze this particular data set. I got it down to 2-3 seconds. A real Erlang person took it over and he got it down to around .8 seconds. I think now the fastest implementation of Tim's system is in something called OCaml, which is another functional language, Python is number two and Erlang is number three. A lot of people say that Erlang can't do file IO and that it's really bad at that, but obviously it must be ok at that because it is pulling in these huge data files and it analyses them on the top ten hits on Tim's website.

InfoQ: Do you see this as something that will continue to happen, that languages become more powerful instead of a general purpose language with a huge set of libraries, tools or middle-ware that sits below it or adds to it? Is this a trend that languages include features that we expect to be in libraries?

Steve: There's a couple of things that are going on in the whole concurrency thing with the multi-core systems ... when you have two cores you can take any old application, throw it on that machine and it's going to do ok. When you have eight cores it gets a little more interesting because you can see that some of them are kind of idle maybe when you run your application. If you don't have the right language to take advantage of that than your applications can use one of the cores. There's nothing the operating system can do to help you because it's not going to take your application and break it up for you. You have to explicitly go in and make it multi-threaded. Threads in languages like Java and C++ are fairly heavy weight. Even though they are lighter than process, they are still heavy-weight. It takes something like Erlang or languages like that they have very, very light-weight threads so it's able to run 50-60.000 threads on my Mac book pro easily. It is a very different style of language.

Then there's also the object-oriented versus functional - and there seems to be a resurgence in functional languages right now. I don't know why that is; it may be because they are so small like you can do so much stuff in just a few lines of code. And even languages like Ruby and Python have functional aspects to them; that may be what's driving it. I think there's a bit of resurgence in language design and people looking at languages. There has always been C as the Assembly language for higher-level languages; not only C++ but others like Python, Perl etc are all built on top of C. There's a lot going on in Java. Java is like the assembling language for the JVM, it becomes the VM for a number of languages, like Scala, Groovy, and Jython. People are moving into these two directions, it's the same direction in fact: building smaller languages better suited to specific problems on top of these general purpose languages underneath.

InfoQ: Of all those languages that you mentioned which one would you recommend?

Steve: I think the past decade or two there has been a search for the language. A lot of people felt C++ was maybe the language that people should be using; then Java came along and a lot of people latched on to Java. I've met many programmers who seem that all they know is Java. If you start recommending to them that maybe they should start looking at other languages some of them get argumentative and they say "Java can do it all!" I think if you were to talk to the people who built these languages they would never claim that their language can do it all. All through this, there have been the multi-language communities that have been rolling along working on these other little languages. Erlang is twenty-one years old, Smalltalk has been around forever and people still use it. I think because of the way that no language can do it all developers really owe it to themselves to learn multiple languages and be able to integrate them.

When you have that choice, when you have a toolbox full of languages and you have a problem and solve it in two lines of Ruby versus two hundred lines of Java it's a really nice feeling. It just makes you a better developer because you start to see how idioms in different languages can be applied and you learn from different languages. In Python there are list comprehensions which are very cool; there's one line that can do all kind of stuff iterating over a list. Erlang has the same thing. You go to Erlang and you say "That's a list comprehension that's almost the same syntactically and does the same things". It's not like every language is a whole different world that you have to completely start from scratch. You learn one, you see some of its idioms, you start to learn another, and you see similar things.

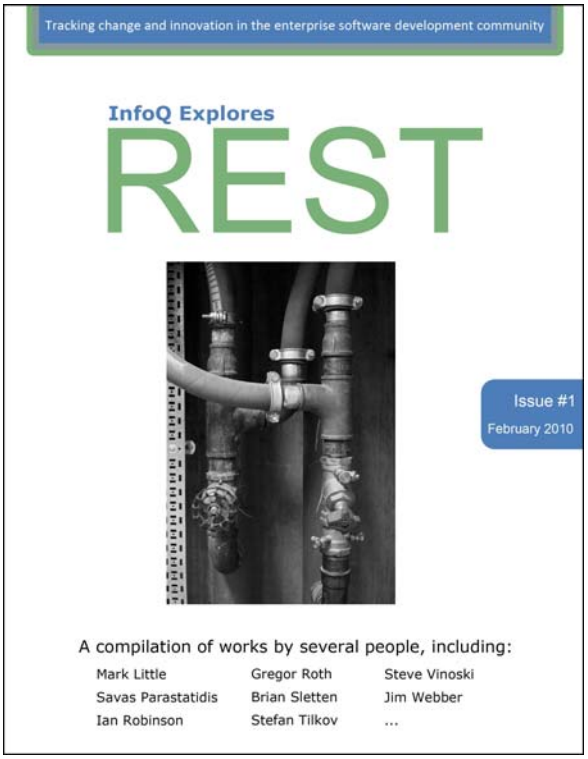
Switching from a OO language to a functional language is going to be a little bit different. Languages like Ruby and Python in particular cross those boundaries and using those you can get a lot of work done and also expand your own horizon at the same time. In terms of concurrency, if you're writing middleware I think you owe it to yourself to look at Erlang. The language itself has the primitives, then there are libraries called the Open Telecom Platform that come with it, that build on those primitives to make reliable software almost simple. It's never simple, but compared to what you have to do, jumping through hoops in other languages, it's kind of a no-brainer. So - there is not one language, look at all of them.

View Full Video：

http://www.infoq.com/interviews/vinoski-gcon-interview

Related Contents：

- How Relevant Is Contract First Development Using Angle Brackets?
- REST – The Good, the Bad and the Ugly
- Quest for True SOA
- Presentation: Scott Davis on Real World Web Services
- EviWare Releases v2.0 of soapUI, a Web Services Test Suite



InfoQ Explores: **REST**

Issue #1, February 2010

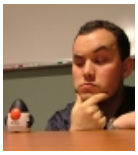
Chief Editor: Ryan Slobojan

Editors:

Feedback: feedback@infoq.com

Submit Articles: article@infoq.com

Cooperation: Cooperation@infoq.com



Chief Editor: Ryan Slobojan

Ryan Slobojan is a managing director at RoundTrip Networks, which focuses on the full lifecycle of online applications including both the application itself and the infrastructure that it runs on. He has worked with a wide range of technologies, but considers Java to be his most in-depth area of knowledge, and has become impressed with the impact that Agile and Lean methodologies have upon the software development process. He enjoys the dual challenges of working with new customers and new technologies, and is constantly scouring the technology landscape for new and interesting technologies which are being used.