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?

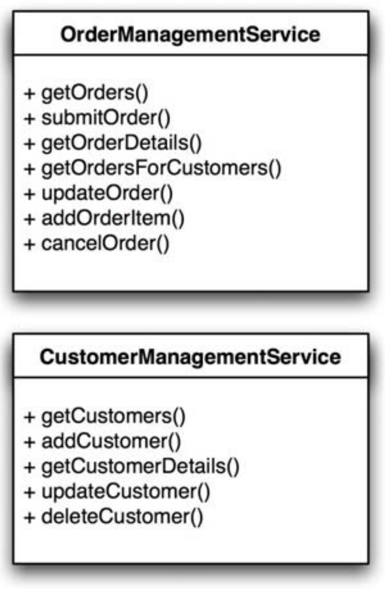
It knows what to do with it because every resource supports the same interface, the same set of methods (or operations, if you prefer). HTTP calls these *verbs*, and in addition to the two everyone knows (GET and POST), the set of standard methods includes PUT, DELETE, HEAD and OPTIONS. The meaning of 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.

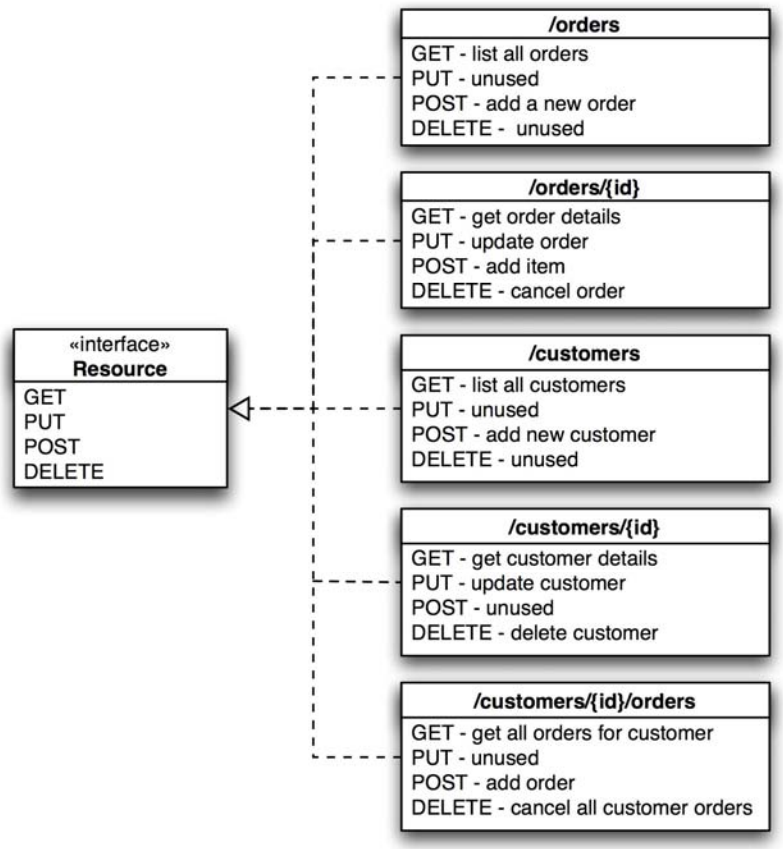
If you expose your application's functionality (or service's functionality, if you prefer) in a RESTful way, *this principle and its restrictions apply to you as well*. This is hard to accept if you're used to a different design approach — after all, you're quite likely convinced that *your* application 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:

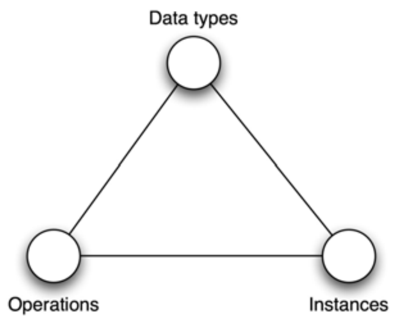


You can see that there are two services defined here (without implying any particular implementation technology). The interface to these services is specific to the task — it's an OrderManagement and CustomerManagement service we are talking about. If a client wants to consume these services, it needs 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

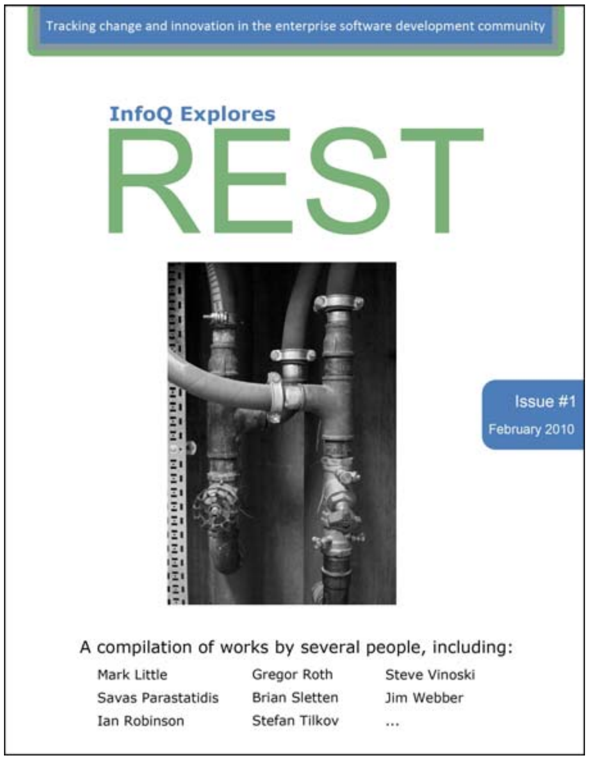
In this article, I have attempted to provide a quick introduction into the concepts behind REST, the architecture of the Web. A RESTful HTTP approach to exposing functionality is different from RPC, Distributed Objects, and Web services; it takes some mind shift to really understand this difference. 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?



Tracking change and innovation in the enterprise software development community

InfoQ Explores

REST

Issue #1
February 2010

A compilation of works by several people, including:

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

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