REST

InfoQueue

eMag Issue 12 - April 2014

What is REST?

In this presentation summary you'll learn precisely what REST is, and is not, about as well as the background of REST that caused it to be defined.

PAGE 9

A BRIEF INTRODUCTION TO REST

Stefan Tilkov provides a pragmatic introduction to REST (REpresentational State Transfer), the architecture behind the World Wide Web.

PAGE 3

HOW TO GET A CUP OF COFFEE

Jim Webber, Savas Parastatidis and Ian Robinson show how to drive an application's flow through the use of hypermedia in a RESTful application.

PAGE 14

REST ANTI-PATTERNS

Stefan Tilkov explains some of the most common anti-patterns found in applications that claim to follow a "RESTful" design and suggests ways to avoid them.

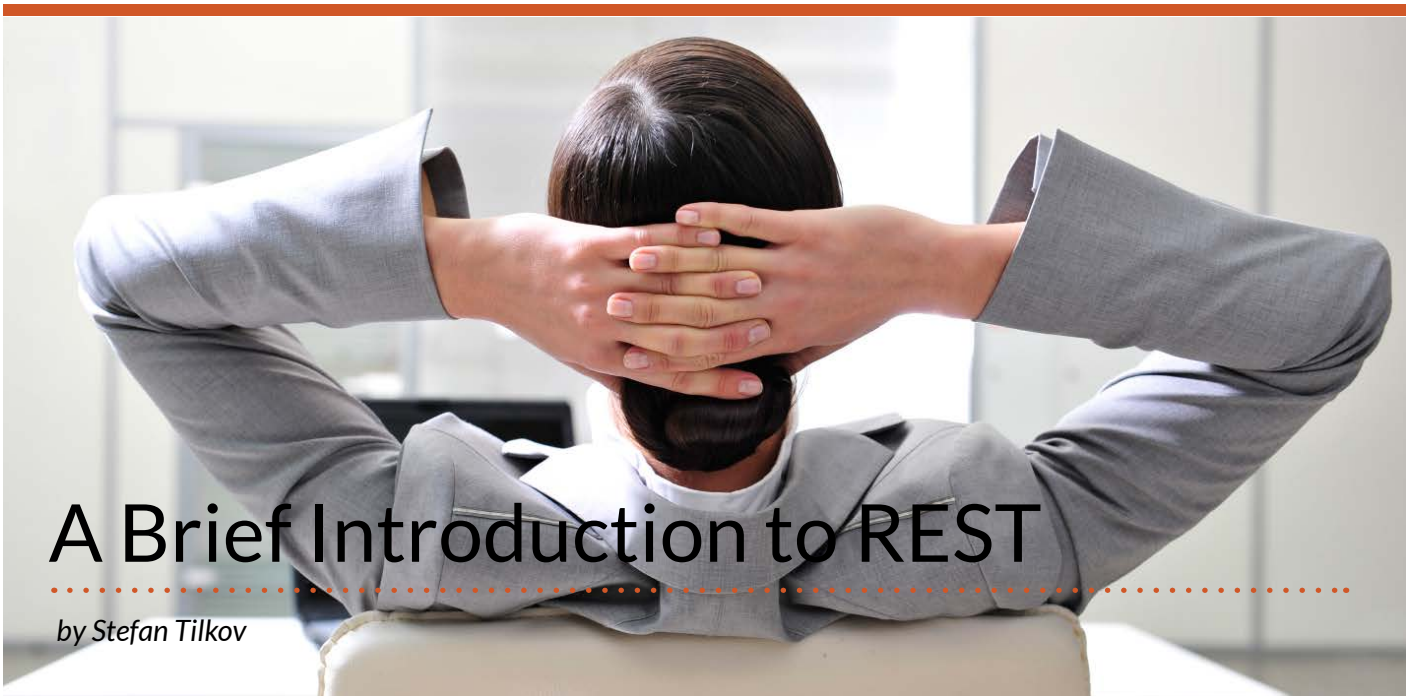
PAGE 28

RESTFUL JAVA EVOLVES

In Bill's session he covers how best to write RESTful services in Java using the JAX-RS standard.

PAGE 33

Contents



A Brief Introduction to REST

by Stefan Tilkov

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 vocal minority claims there's a better way: REST, short for REpresentational State Transfer. This article will try to provide a pragmatic introduction to REST and RESTful HTTP application integration without digressing into this debate.

Key REST principles

Most introductions to REST start with the formal definition and background. A simplified, pragmatic definition of 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. In the systems that people build, there is usually a set of key abstractions that merit identification. Everything that should be identifiable should obviously get an ID, and 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. Consider an arbitrary high-level object within the last application you built (assuming it wasn't built in a RESTful way): it is likely that you would have profited from this in many use cases. 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. 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 persistence aspects as an implementation detail. But this is not a conflict at all. Usually, the things — the resources — that merit 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. The idea of identifying everything that is worth being identified 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 (assume 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 identification, 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; for the moment, just keep in mind that links are an extremely useful way to make an application dynamic.

To summarize this principle: 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?

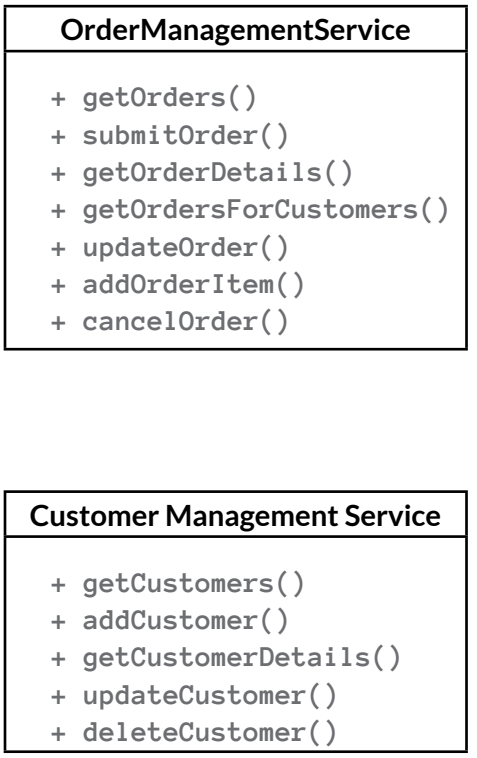
It knows what to do with it because every resource supports the same interface, the same set of methods (or operations, if you prefer). HTTP calls these verbs, and in addition to the two everyone knows (GET and POST), the set of standard methods includes PUT, DELETE, HEAD, and OPTIONS. The meaning of 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 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 likely convinced that your application has more logic than is expressible with a handful operations. Let me spend some time trying to convince you that this is not the case.

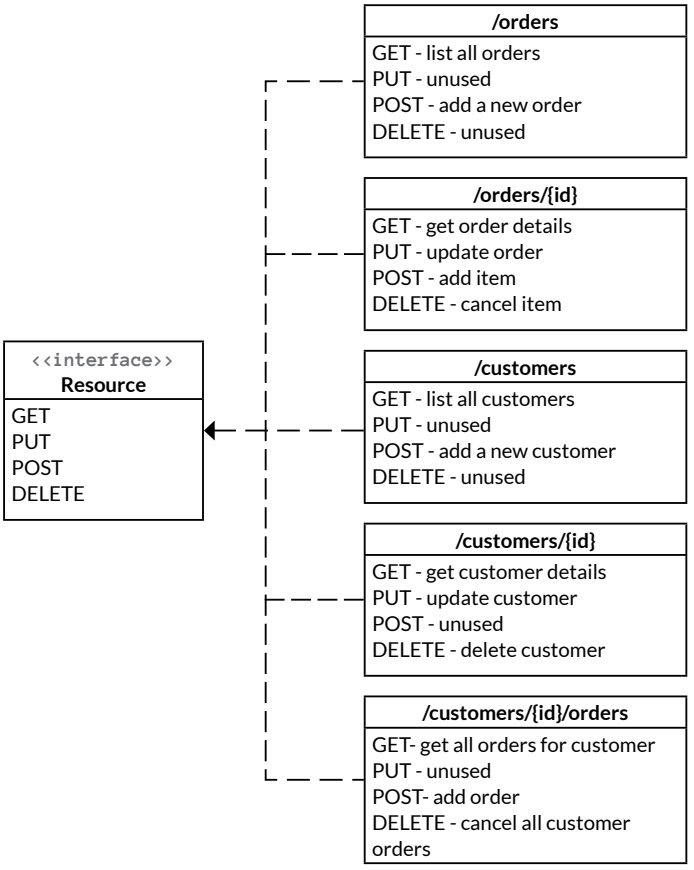
Consider the following simple procurement scenario:



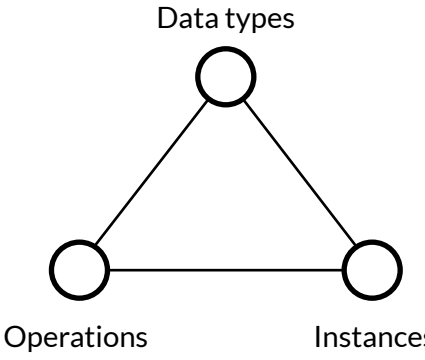
You can see that there are two services defined here (without implying any particular implementation technology). The interface to these services is specific to the task — it's an OrderManagement and a CustomerManagement service we are talking about. If a client wants to consume these services, it needs 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 upon which to invoke those fixed methods. The point of this is to illustrate that you can basically express anything you like with both approaches.

Essentially, this 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 to the Web a few million customer URIs. If it's designed the 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 resources only to 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. Say 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 the representations of a resource should ideally 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 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.

Another way to exploit this is to 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 redesign: you can't simply stick a URI to some session state and call it RESTful.)

But other aspects might be 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 differently than usual, so I did not provide the formal background and history of REST at the beginning. Let me try to address this, if somewhat briefly.

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 look at the history of REST.

The term REST was defined by Roy T. Fielding in his Ph.D. thesis (you might actually want to follow that link — it's quite readable for a dissertation). Roy had been one of the primary designers 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. 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 with 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 Roy defined the REST style after the Web — 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. As Roy is both the author of the REST dissertation and a strong influence on the Web architecture's evolution, 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. With some justification, one can say that using HTTP without following the REST principles is abusing HTTP. 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, applications violate REST constraints 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). More on this and other notable abuses will come in a follow-up article.

Summary

I have attempted to quickly introduce 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. Awareness of REST principles is beneficial whether you are building applications that expose only a Web UI or want to turn your application API into a good Web citizen.

ABOUT THE AUTHOR

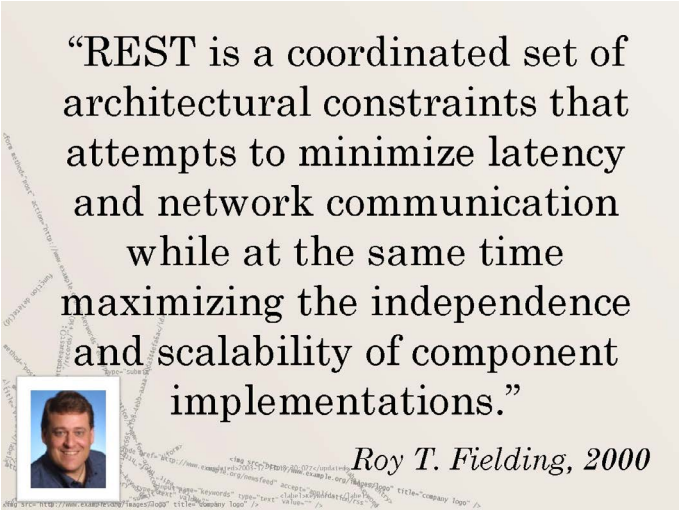
Stefan Tilkov is co-founder and principal consultant at innoQ, where he spends his time alternating between advising customers on new technologies and taking the blame from his co-workers for doing so. He is a frequent speaker at conferences and author of numerous articles. Twitter: @stilkov

READ THIS ARTICLE ONLINE ON InfoQ

What Is REST?

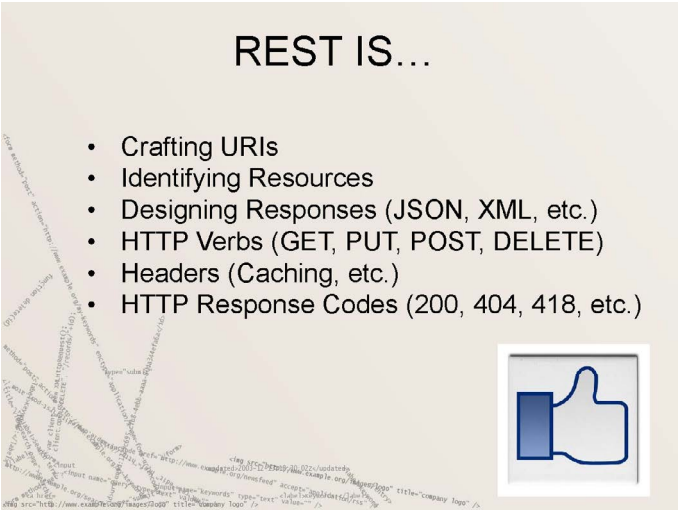
by Mike Amundsen, Director of API Architecture, API Academy, CA / Layer 7 Technologie

Roy T. Fielding described REST more than a decade ago in a dissertation on the styles and design of software for networks. The REST chapter was just one in a six-chapter dissertation; about 25 pages of a 172-page document. REST is an example of how Fielding thought about establishing network-level architecture. It's not the ultimate; it's not the only. It's just the example of analysis and design of a style.

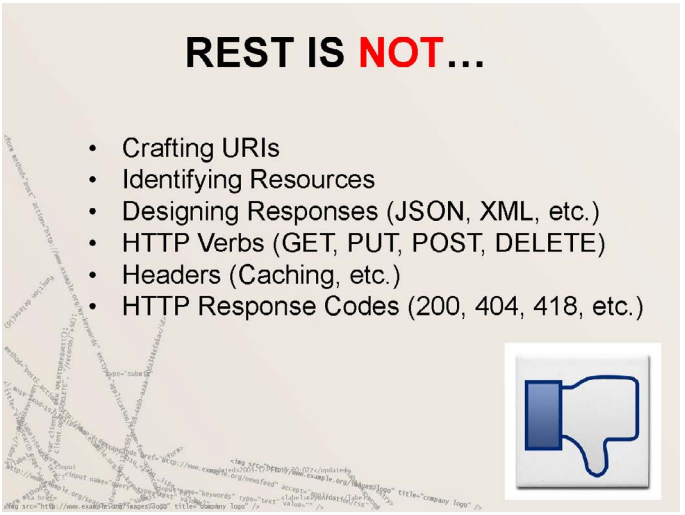


Fielding's coordinated set of architectural constraints attempts to minimize latency and network communication while at the same time maximizing independence and scalability of component implementations. So, REST attempts to minimize latency and maximize scalability. That's a simple goal, and it's a goal that a lot of us have, no matter what we're doing.

However, sometimes when we hear "REST", we hear something very different.



So very often when people talk about REST, they talk about the implementation details of making sure your objects or your URLs or whether messages have curly braces or angle brackets or or whether you are using the right verbs, and so forth. The problem is: that's not right. That's not REST.



So what is REST? It's a set of constraints. It's not even a set of rules, more like guidelines, actually. REST defines constraints for developing software that exsits on an network. It's a way to identify constraints with which we can successfully build software that works around the world. Let's make it possible for people build software that works together even when these people don't know each other; when they don't speak the same language[when they don't even use the same software products. Fielding uses this notion of identifying constraints so that we can minimize latency and maximize scalability at a network scale, at a planetary scale.

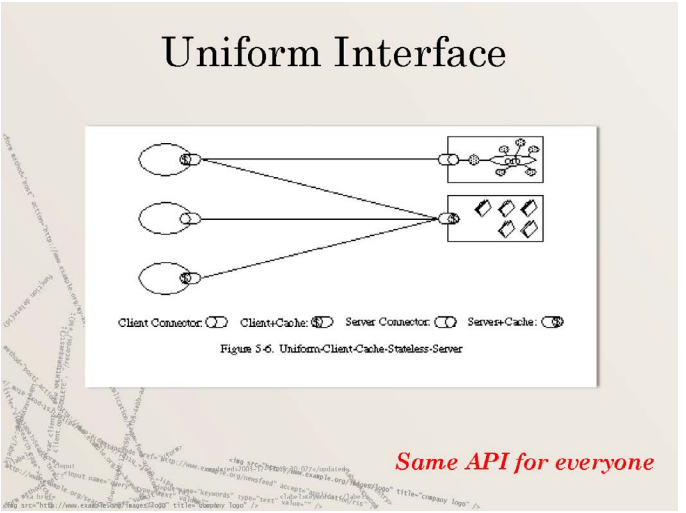
REST Constraints

Fielding identifies six constraints. The first one is the idea of the "client-server" relatoinship. The client always initiates; the server always responds. That means no peer-to-peer implementations. Fielding thinks, in order to meet his goals about maximizing and minimizing, that client-server is the constraint that should be adopted by everyone.

The next constratinis "statelessness". Fielding writes in his dissertation that the server should not be responsible for remembering every party to which it is talking. Servers should be able to forget a client instantly. The next time something comes in, the server should be able to start afresh.

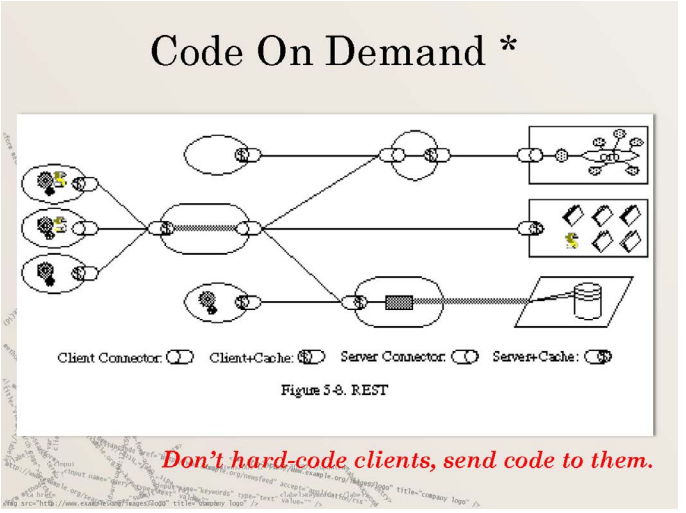
Then there's the "caching" constraint. Caching means essentially "I don't want to hit the origin server every time I make a request" Why, because if every client app has to speak directly to the server every time then you won't be able to scale. See, the number of servers really shot up in the early life of the Web -- in the late '90s and early 2000s. The notion of caching was essential to making sure that the Web continued to work correctly as the ratio of clients to servers grew astronomically. The Web works because we have caching. The constraint tells us we need to design a system that supports this kind of caching.

Now, here's the big one. Fielding decided there should be a "uniform interface": that everyone on the Web had to use the exact same API. This was radical. Thismeans that everyone has to use the same programming interface, no matter the application – whether you're doing a webpage, an accounting application, or a drawing application you all have to use the same exact API. This is actually the hardest constraint to deal with.

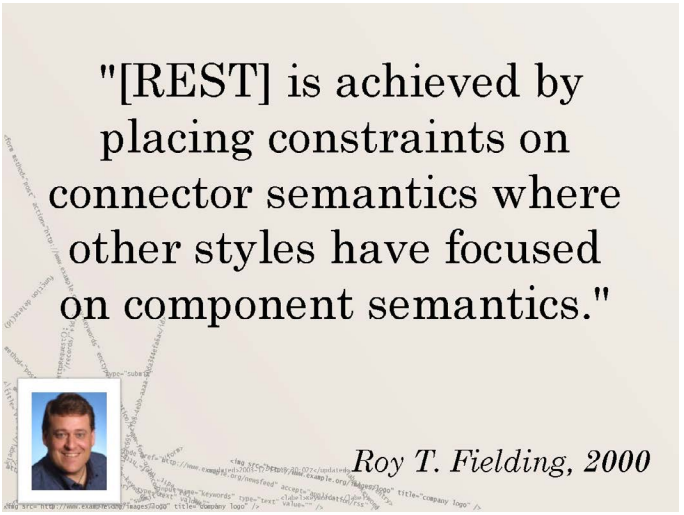


The fifth constraint is the idea of a "layered system": the infrastructure should work at a planetary scale so you can add or remove hardware at any time without any downtime. The Web has to be up 24/7. It has to be ready to grow along the way. We have to be able to add new hardware even 10 years from now that still works with things written today or even ten years ago.

Finally, there's "code on demand".



How can I hard-code clients to run an application that hasn't been invented yet, especially if I'm going to limit the interface so that everybody has to use the same one? To do that you send code to the client and let the code run on the client. That's why we have JavaScript. Originally, this idea of code on demand applied to Java applets. And Microsoft had ActiveX. But it turns out Javascript is the most common way we implement this constraint today.



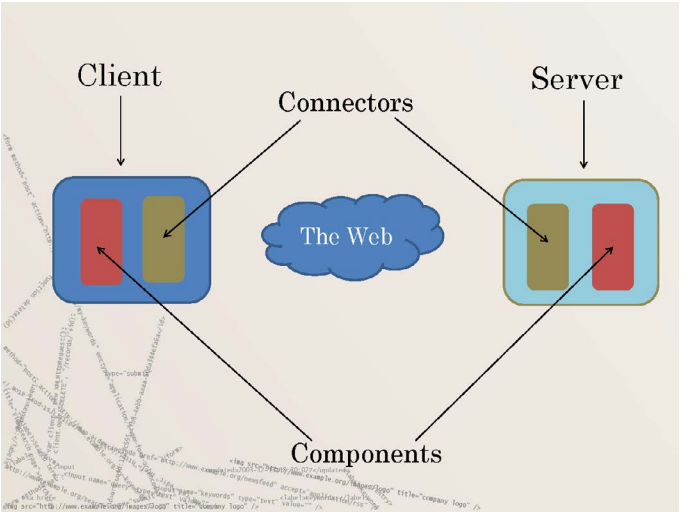
Connectors and Components

Fielding wanted to constrain the way things *connect to each other*, not the way they operate. To do that, he highlights this notion of connector and component. The connector is not the same as the component. It's important for us to keep them separated. Very few of us really think about this when we're building large-scale applications.

What are components? We can identify lots of components: databases; file systems; message queues; transaction-manager tools; all your source code. These are components. These are things that are specific to the problems you're trying to solve, and each operates in a specific way. Even different databases do things in unique ways. MySQL does things differently than the Microsoft's SQL server. CouchDB does things differently than MongoDB. These components all unique. Each solves problems in a particular way.

It's the same for file systems, whether Unix-based, Windows-based, or Mac-based. At he network level, we don't really care. It's just a local file system. And the way you ***queue*** information, the way you decide when a transaction ends and when it stops, and so forth --all these things are entirely up to you when you select your components. These are your components: your operating system, your tools, your programming languages. But what's really important is they're *private*. In otherwords, components are your "private parts"and it's best to keep them to yourself. I don't care what database you're using. You can change it anytime you want to because your database is private – it's hidden -- and we can't see it on the Internet.

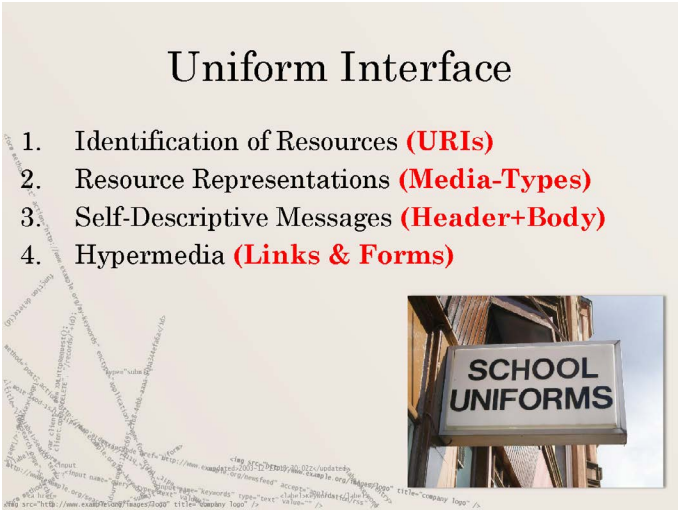
That is way Fielding wants us to use the same API for the *connectors*, not the components. Connectors include Web servers, browser agents like your typical browser, proxy servers, caching servers, etc.. These connectors are the things that understand how that one API works. And, on the Web, that one API is most often the HTTP protocol.



When you build an application, you don't build connectors; you use them. Every major language has HTTP connectors and caching connectors in the form of classes or libraries. You don't have to rewrite that. The connectors are what we all agree upon ahead of time. We have to agree upon upon them because it's all about the Internet. Once we agree upon the connector API then, you can use whatever programming language you want, you can be as creative or as mundane or as loose or strict or whatever it is that you want to be, as long as you support the same connectors that everyone else uses..

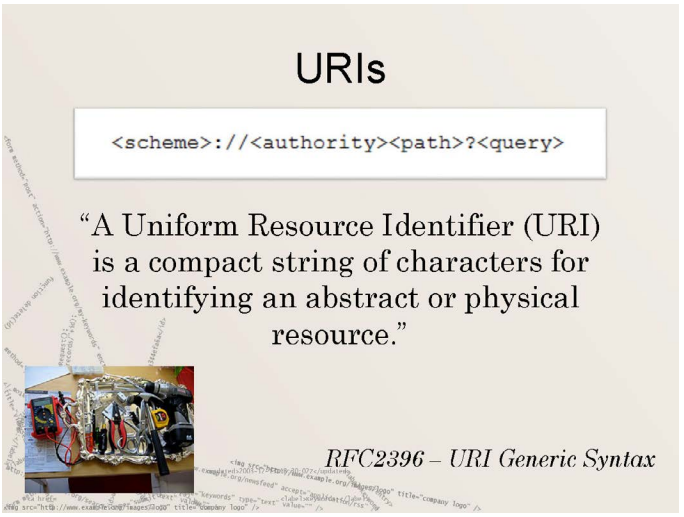
Of course, all clients and servers have their own private components. They do whatever is important in whatever manner they wish whether it's manipulating images, storing data, or computing things. That's all specific to that component. but we all share the idea of connectors like the ones fo HTTP or FTP or IRC or DNS or SMTP and so forth. We have this whole series of standards in order to make sure we have shared connectors.

How do I make a connector model work when I am creating a unique applicatoin? How do I make scalable applications work if I'm talking over generic connectors? That brings us back to Fielding's uniform interface.



The Uniform Interface

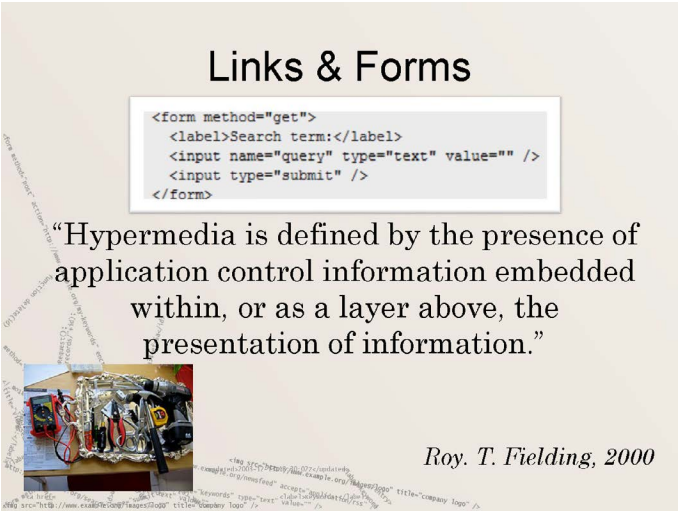
See, there are four things inside Fielding's version of a uniform interface that make this all work. First, there is a way to **identify things**. in REST we use URIs. URI is a general name for things like URLs and URNs. And URIs are pretty cool:



Second, we use media types as ways to **represent information**. HTTP allows us to represent the same thing in lots of ways, such as an HTML table, an CSV file, or a PNG image. But media types are not always well understood, partly because they were not invented for the Web but for email as the MIME – the Mail Interchange Message Extension. The people who invented HTTP wanted to have a variable representation model so they just took the one they had available at the time - MIME. But what makes media types on the Web different than email is that media types identify not only the nature of the data but also some additional information. In fact, on the Web, media types can also identify processing rules for the data.

Then there is number three: **self-descriptive messages**. This is an odd phrase that means everything you need to understand the message is in the message itself. In HTTP we use the headers as well as the body of the message to create a self-descriptive package. A package that includes not just the content, but all the stuff *about* the content such as where I got it from; whether it's cached; whether it's a stale representation; how long you can keep it before it goes stale; what format it's supposed to be; whether or not it the request made for this data was authenticated; and more.

In fact, heaers are also kinda cool in HTTP. There are headers for the request that ask for a specific version or representation. There are headers for response. The server says, "Well, I couldn't really give you that representation, but I gave you this one instead." And there are headers for the content itself, like what spoken language it contains, whether or not it's compressed data, or how many bytes are in it.



Hypermedia is application-control information. It's a way to control aspects of the application across the world from one place to another by sending those hypermedia controls.

Fielding's REST

As dissertations go, Fielding's is eminently readable. Chapter 5 is the one that talks about REST. Chapters 2 and 3 are much, much better, I think. Chapters 2 and 3 are sort of historical. In those two chapters, Fielding talks about what he thinks should be important in a network architecture, and he talks

about other people's versions of this, too.. Chapter 2 is a great opportunity to look at other visions of network systems based on things like peer-to-peer architectures, event-based architectures, and RPC style architectures. Reading the others chapters in Fielding's dissertatoin puts REST in perspective.

Ultimately, I think one of the most interesting ideas in Fielding's dissertation is the notion of uniform interface for applications on the network. It's a complicated idea. But it works because it's based on the simple principles of identification, representation, self-descriptiveness, and hypermedia. This provides a kind of network-level programmability that relies on the fact that the message can tell you what you can do next and, at the heart of it, is based on his other provocative idea of connectors. Connectors are important because we can count on them to "just work". hen we can concentrate on our components. We can be creative with our unique components and be assured they will work with the shared connectors.

When we do that, we have a chance to build things that last. That's really all any of us want to do, I think. REST is just one way to talk about this idea of building things that will last, one way to do it. Occassionally someone will look at some Web implementation and claim , "That's not RESTful," and then people will argue about it for a while.. Frankly, I don't care if an implementation is RESTful or not as long as it solves the problem that needs to be solved. So, if I use some of these constraints and not others; if I add new constratins that aren't listed in Fielding's dissertation, that's because I have different needs and I have different goals. And, in the end, Fielding's dissertation is about picking constraints consciously and proactively in order to come up with the best architecture for the problem at hand. And who would argue about that?

ABOUT THE SPEAKER

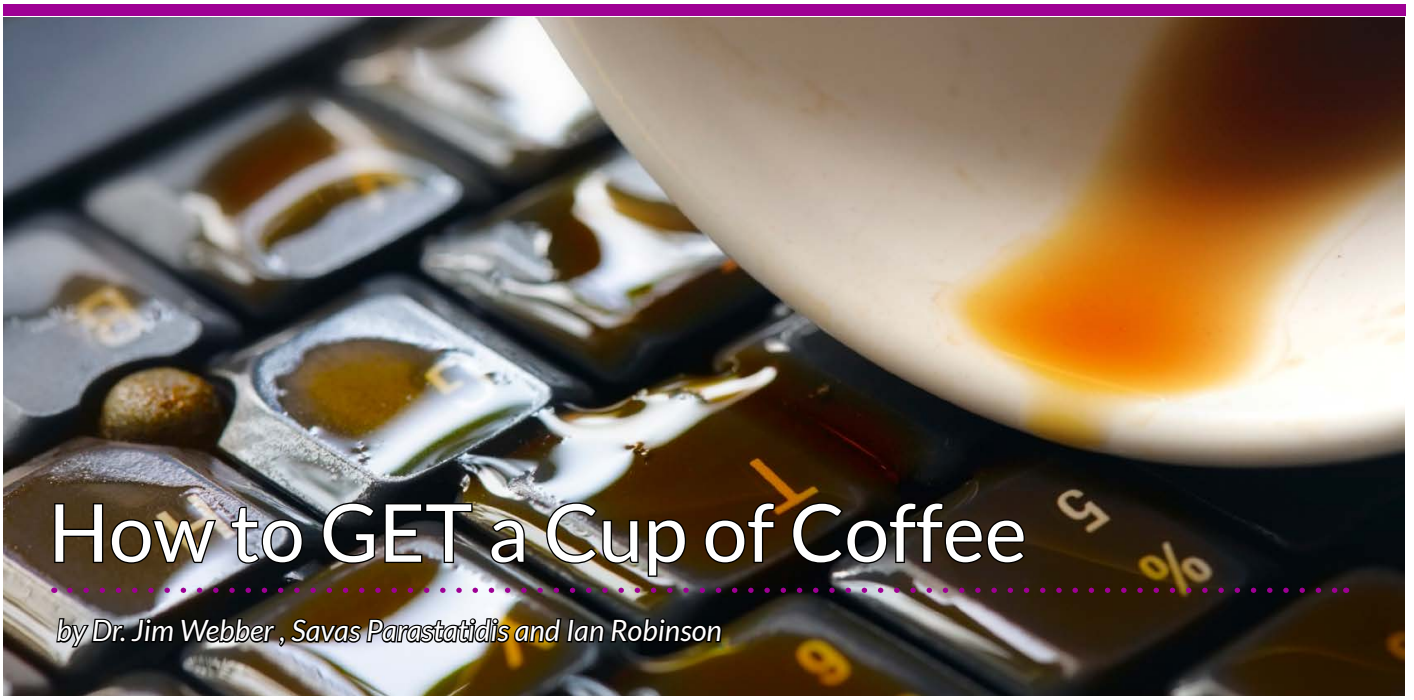
An internationally known author and lecturer, **Mike Amundsen** travels throughout the world consulting and speaking on a wide range of topics including distributed network architecture, Web application development, and other subjects.

In his role of Director of Architecture for the API Academy, Amundsen heads up the API Architecture and Design Practice in North America. He is responsible for working with companies to provide insight on how best to capitalize on the myriad opportunities APIs present to both consumers and the enterprise.

Amundsen has authored numerous books and papers on programming over the last 15 years. His most recent book is a collaboration with Leonard Richardson titled "RESTful Web APIs". His 2011 book, "Building Hypermedia APIs with HTML5 and Node", is an oft-cited reference on building adaptable distributed systems.Please use content from http://g.mamund.com/shortbio to build the speaker bio.

Presentation transcript edited by Mike Amundsen and Mark Little

WATCH THE FULL PRESENTATION ONLINE ON InfoQ



How to GET a Cup of Coffee

by Dr. Jim Webber , Savas Parastatidis and Ian Robinson

We are used to building distributed systems on top of large middleware platforms like those implementing CORBA, the Web services protocol 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 through a simple customer-service 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 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.

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 that a workflow implements 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 that 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 apologise in advance for taking liberties with the way Starbucks works because our goal here isn't to accurately model Starbucks but to illustrate workflows with Web-based services. 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 (figure 1) and the other captures the set of actions performed by a barista (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.

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 prepare, making 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.

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 acting on a resource via its URI, causing state changes.

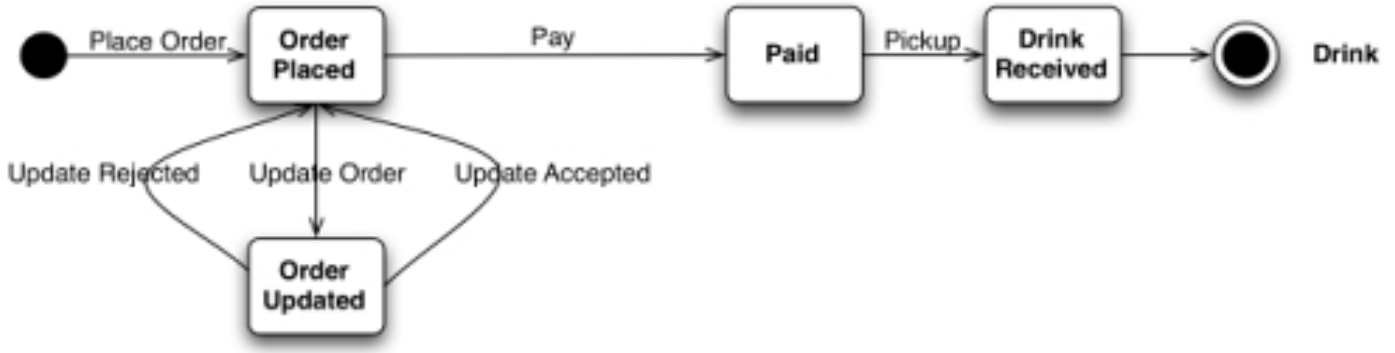


Figure 1: The Customer state machine

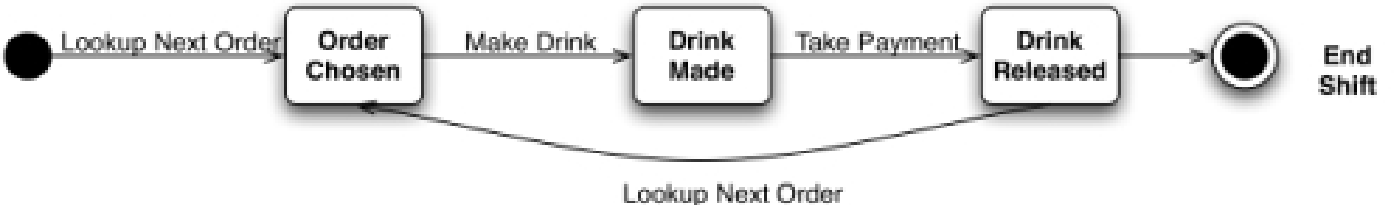


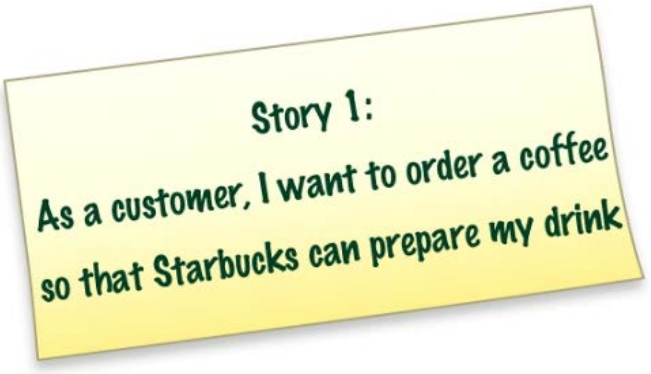
Figure 2: The barista's state machine

„GET and HEAD are special cases because 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. Notions of state machines and the Web aren'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

A simple story card kickstarts the 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) that 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.

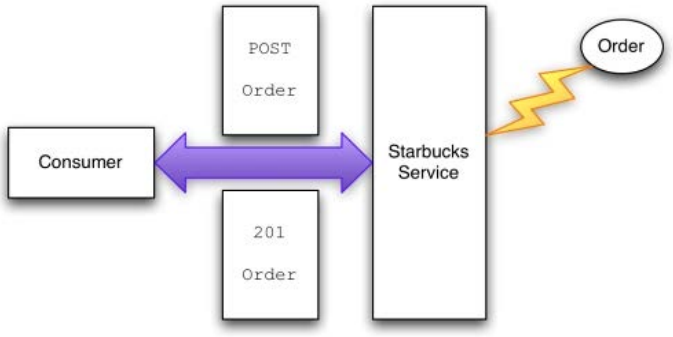


Figure 3: Ordering a coffee

Figure 3 shows the interaction of placing an order with Starbucks. Starbucks uses an XML dialect to represent entities from its domain. Interestingly, this dialect 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 drink order

The Starbucks service creates an order resource, 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 confirms 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 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. The <next/> tag here is meaningful in the context of the customer workflow, the semantics of which have been agreed upon 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 too lazy (or secure) to give a 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.

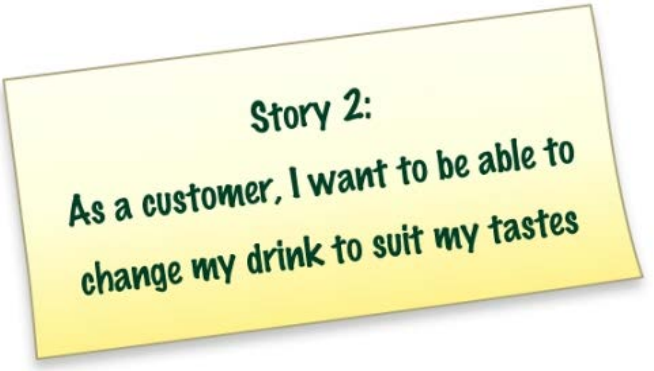
`412 Precondition Failed` - The request wasn't processed because an ETag, If-Match or similar guard header failed evaluation. We need to figure out how to progress.

`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 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: anyone that really likes coffee will not appreciate a single shot of espresso swamped by a pint of hot milk. We're going to have to change that. Fortunately, the Web (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 the order – 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`

Figure 6: Asking for OPTIONS

In figure 6, we see that the resource is readable (it supports GET) and 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!

Had it 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 that contains 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 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 in which 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 (one-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 travel a happy path and assume we managed to get our additional shot of espresso. Of course, Starbucks won't hand us 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. 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 upon 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 in 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 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 on 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 upon upfront.

Consumers typically agree on 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 cost of our order from the <cost> element embedded in the order representation, so our next step is to send a payment to Starbucks to prompt the barista to 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 authentication. We'll see how authentication works from Starbucks' point of view later.

```
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

The exchange shown in figure 12 is all we need for a successful payment. 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 don't go wrong or be recoverable when they do2. We can predict a number of potential problems 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. In 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 has in fact succeeded (effectively an acknowledgement of a NOP 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 has responded with a 500, 503, or 504 response code.

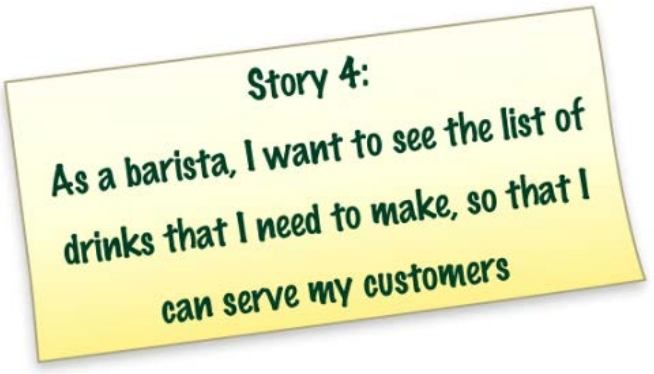
Status codes in the 4xx range are trickier, but they still indicate how to move forward. For example, a 400 response indicates that we PUT something the server doesn't understand, and we should correct our payload before PUTing it again. Conversely, a 403 response indicates that the server understood 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 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 for 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 to 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 in the shop. 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:starbucks: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 updates frequently, so the barista will need to poll it to stay up to date. Polling is normally thought to offer low scalability; the Web, however, supports an extremely scalable polling mechanism, as we'll see shortly. And with the sheer volume of coffees Starbucks manufactures 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 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.

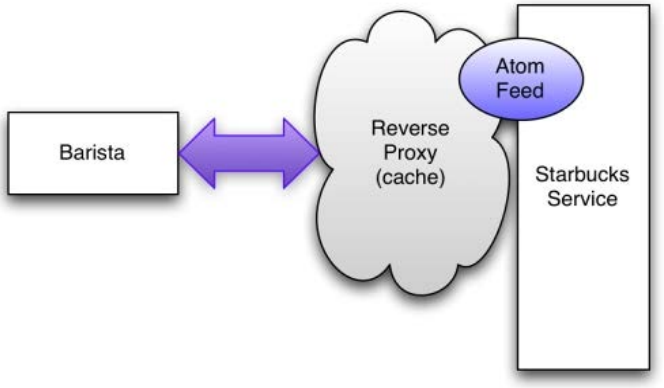


Figure 14: Caching for scalability

For most resources – especially those that are accessed widely, like our Atom feed for drinks – it makes sense to cache them outside 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 server failures and helps crash-recovery scenarios by improving the availability of the 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

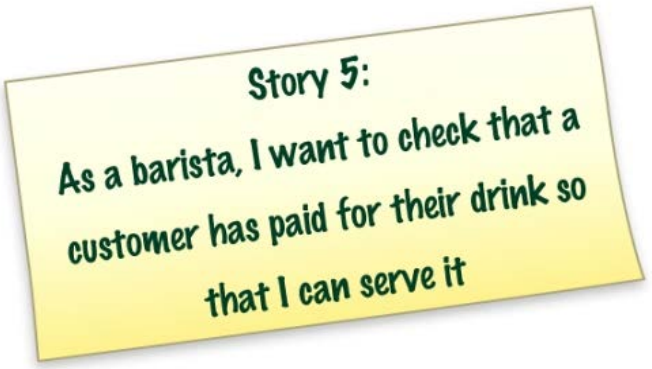
cached orders are cleared, the Starbucks service uses the Expires header to declare how long a response can be cached. Any caches between the consumer and service (should) honour that directive and refuse to serve stale orders3, instead forwarding the request to the Starbucks service, which has up-to-date 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 six requests per minute, with the remainder handled by the cache infrastructure. Even for a relatively poorly performing service, six 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), Web-based solutions are not a great idea. If, however, you can accept latency on the order of seconds, or even minutes or hours, 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 to forbid any further updates. 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 figures 6-10).



Fortunately, we can use a well-defined protocol 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.

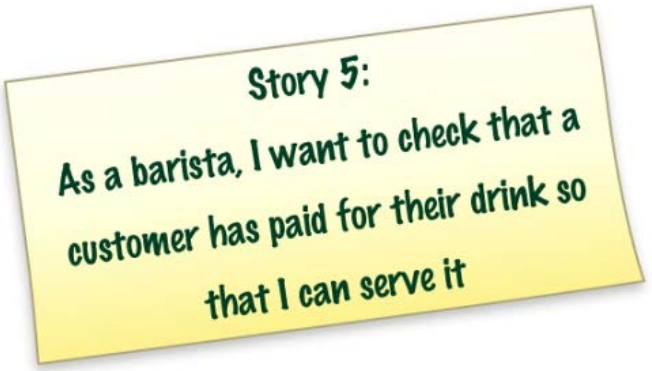
```
PUT/order/1234 HTTP 1.1
Host: starbucks.example.com
Content-Type: application/atom+xml
Content-Length: …

<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 needs to know that we've paid for the order before they release the coffee to us. 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: Challenging unauthorised access to a payment resource

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 individually to each order resource. 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 on 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 that provide state transitions to the offer. Thanks to the magic of URIs, the links may lead to a third-party partner just as easily as they could 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 to 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 concludes 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 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 forcing us to invent new exceptions or faults: HTTP provided everything we needed right out of the box. And even on 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 unleashes 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.

---

1. An ETag (an abbreviation of entity tag) is a unique identifier for the state of a resource. An ETag for a resource is typically an MD5 checksum or SHA1 hash of that resource's data.

2. When safety is at stake, of course, we prevent things from going too far wrong in the first place! But receiving coffee isn't a safety critical task, even though it might seem that way for some most mornings!

3. HTTP 1.1 offers some useful request directives, including max-age, max-stale, and max-fresh, which allow the client to indicate the level of staleness it is prepared to accept from a cache.

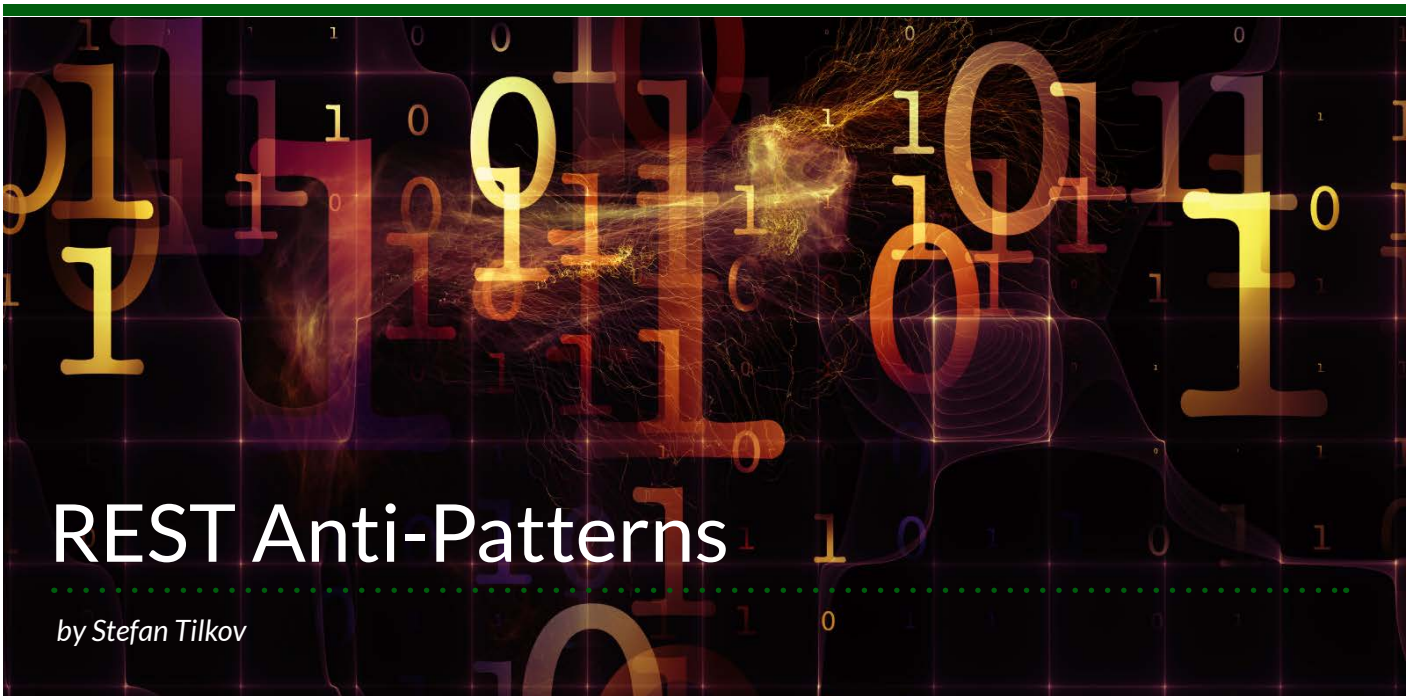
READ THIS ARTICLE ONLINE ON InfoQ

ABOUT THE AUTHORS

Dr. Jim Webber is chief scientist with Neo Technology, the company behind the popular open-source graph database Neo4j, where he researches and develops distributed graph databases and writes open-source software. His is a co-author of the book REST in Practice. Jim's blog is located at http://jimwebber.org and he tweets often @jimwebber.

Savas Parastatidis is a software architect in the Online Services Division at Microsoft. He builds technologies for reasoning over the world's information and is interested in all things related to semantic computing, knowledge representation and reasoning, the Web, and large-scale services, which led him to co-author the book REST in Practice.

Ian Robinson is an engineer at Neo Technology, currently working on research and development for future versions of the Neo4j graph database. He is a co-author of REST in Practice (O'Reilly) and a contributor to REST: From Research to Practice (Springer) and Service Design Patterns (Addison-Wesley).



REST Anti-Patterns

by Stefan Tilkov

When people start trying out REST, they usually start looking 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 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 ways are in line with what the Web's designers had in mind but many are 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 a 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. I should be careful to distinguish "REST" from "RESTful HTTP". I'm not careful, so let's 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, and the use of syndication to model events.

In this article, 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=del
eteCustomer&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 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 meant to be bookmarked.
4. There is a risk that link-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 it could 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, 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 similar to the first one, only this uses the POST HTTP method. POST carries an entity body, not just a URI. A typical scenario uses a single URI to POST to, with 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 the SOAP message, possibly including some WS-Addressing SOAP headers, that determines what happens.

One could argue that tunneling everything through POST shares all the problems of the GET variant; it's just a little harder to use and can neither explore caching (not even accidentally) nor support bookmarking. It doesn't actually violate any REST principles as much as 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 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, 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 something's 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 exploit the available caching

capabilities to avoid unnecessarily retrieving a representation a second time. 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 during 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 good guide, turn to Mark Nottingham's Cache Tutorial.

Ignoring status codes

Unknown to many Web developers, HTTP has a 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") errors. 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 only 200 (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? Not necessarily. One of the key ideas of REST is statelessness, although not in the sense that a server cannot 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 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 having 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 there are 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 simply because the server 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. Ideally, a client should have to know only a single URI.

Everything else – individual URIs and 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 that 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 beforehand 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 while totally forgetting the hypermedia aspect. My advice is to limit the time you spend on finding the perfect URI design (after all, they're 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 respectively implemented in Java, JavaScript, and Ruby. 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 only one representation format, it often indicates 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, predefined, widely-known formats than to invent proprietary ones – an argument that leads to the last anti-pattern addressed in this article.

Breaking self-descriptiveness

Breaking the constraint of self-descriptiveness is an anti-pattern so common that it's visible in almost every REST application, even in those created by those who call themselves "RESTafarians", myself included. Ideally, a message – an HTTP request or HTTP response, including headers and the body – should contain enough information to allow any generic client, server, or intermediary 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 your system to start its default PDF viewer; and finally you can read the PDF on your screen. All other users in the world could use their 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, you can consider anything not standardized by an official standards body to be breaking this constraint, and can be considered a case of this anti-pattern. In practice, you strive to follow 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 Design Patterns: Elements of Reusable Object-Oriented Software, which kickstarted the patterns movement, many people have 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 to violate any rule, or, in REST terminology, to relax any particular constraint. It’s fine to do so, but it’s useful to be aware of the fact, and make a more informed decision.

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

ABOUT THE AUTHOR

Stefan Tilkov is co-founder and principal consultant at innoQ, where he spends his time alternating between advising customers on new technologies and taking the blame from his co-workers for doing so. He is a frequent speaker at conferences and author of numerous articles. Twitter: @stilkov

READ THIS ARTICLE ONLINE ON InfoQ

RESTful Java Evolves

by Bill Burke

JAX-RS is a server-side annotation framework and is a required API for Java EE6. If you download a compliant application server you will have a JAX-RS implementation within it and integrated with other component layers that are provided with enterprise Java. It allows you to use annotations to map incoming HTTP requests to Java method calls. It's actually maps between requests and Java objects, so you can reroute the request to a specific Java method and Java object.

What are JAX-RS's strengths? For one thing, it's annotation-based, which gives developers a DSL for mapping an HTTP request through a Java method. It's a nicer way of doing servlets and a simple way to map an HTTP request to a Java method. It doesn't impose any specific implementation architecture on you, unlike some other RESTful frameworks out there which conflict with the way enterprise Java developers are used to writing systems: a company that's building a RESTful Web service within their internet are going to be interacting with pre-exisitng business logic. They're going to be interacting with databases. One of the strengths of JAX-RS is that it gives them a familiar implementation strategy to work with.

Perhaps JAX-RS's biggest weakness is that it has little hypermedia support. This is a big deal: hypermedia is the key to writing strong RESTful services. JAX-RS does have some helper APIs so you can build and parse URI templates, but beyond that it doesn't have anything standardized. Each implementation-specific framework does have its own way to inject links into an XML JAXB document object model etc., so they have proprietary ways of handling hypermedia support.

What does JAX-RS look like for the developer?

```
@Path("/orders")
public class OrderResource {
  @GET
  @Path("/{order-id}")
  @Produces("application/xml",
"application/json")
  Order getOrder(@PathParam("order-id")
int id,
{
... }
}
```

Assume you're writing a Java class. You need to map an HTTP request to an actual Java method call. You define path expressions using the Path annotation so that you can match the request URL to a particular Java method. The getOrder method will service a GET request and the Path annotation defines the URI expression to match the GET annotation.

The Produces annotation defines the media types returned by this GET request. JAX-RS also supports content negotiation so if a client sends an Accept header and wants JSON back, this method will automatically return JSON to the client if there is a corresponding marshaller.

The getOrder method returns a domain class called Order. JAX-RS will take this Order object and automatically marshal it into a specific media type: XML; JSON; or any other marshaller that you can plug in. And then there are a la carte injections where the class can tell JAX-RS that it is interested in specific parts of the HTTP request, for instance, the PathParam, HeaderParam, and QueryParam annotations.

You can plug in your own marshallers and unmarshallers. There's a complex response object that will let you return your own headers, do allocation forward, or send back a specific complex response. It also has exception mappers so if your services throw a type Java exception you can plug exception mappers to automatically catch those exceptions and send back an appropriate HTTP response. And there's all the other APIs that help you do conditional PUTs and POSTs and that sort of thing.

There's also asynchronous HTTP. Several years ago, when AJAX came along, often servers needed to push information to the clients with which they interacted – e.g., a stock-quote server at a financial institution had to push the next available quote to the client. These clients would connect through browsers using AJAX, block on the server with a GET request, and wait indefinitely until the server had to update their information. This architecture resulted in thousands of GET requests blocking and thousands of threads being consumed at the same time. The Linux-kernel threads were heavyweight back then. They ate up a lot of stack space per thread so they couldn't handle thousands of threads at one time.

This is where asynchronous HTTP comes in. You can have one specific thread handle responses from multiple clients. JAX-RS implements this with an ExecutionContext interface similar to the async context in Servlet 3.0. It allows you to manually suspend an HTTP request and manually resume an HTTP response in a separate thread.

Assume you have a QueueResource and you have a client that wants to poll that particular QueueResource and wait for a response. You could use the Suspend annotation on a method that's annotated with GET; inject an ExecutionContext and all you have to do is queue that particular ExecutionContext so that a different thread could handle the response back.

```
@Path("/queue")
public class QueueResource {
  @GET
  @Suspend
  @Produces("application/json")
  public void pull(@Context
ExecutionContext ctx)
  {
    queue.put(ctx);
  }
}
```

The client API for JAX-RS is a fluent low-level API: a pattern of writing your Java APIs so they look like DSLs. With this API, you can reuse the content handlers, the message-body readers, and writers that you originally wrote on the server side. You can reuse them back in the client side.

There are really two major pieces of the client API. You have the client interface, which deals with configuration, connection management, etc., and a WebTarget interface that represents a particular URI. It's really an abstraction for a URI or a URI template. And from that WebTarget, you can build requests.

One good thing about this WebTarget is that different services you interact with might have different authentication models. One URL might use digest authentication while another might use basic authentication. You can differently configure each WebTarget you create.

```
WebTarget uri =
         client.target("http://shop.com/
orders/{id}");
uri.register(new CookieCache());
uri.register(new ResponseCache());
uri.setProperty(USERNAME, "bill");
uri.setProperty(PASSWORD, "geheim");
uri.setProperty(AUTH_TYPE, BASIC);
```

The client API also has an asynchronous interface. It's really two different forms. You have a feature from Java UDL, the concurrent package that's in Java, and you have the ability to register a callback interface. It looks the same as a synchronous request except you are inserting async in there. In this particular case, instead of getting back an order object, we're getting back a Future that we can pull to see if the request has been done or not.

```
Future<Order> future =
          client.target("http://shop.
com")
                .request()
                .async()
                .get(Order.class);
Order order = future.get();
```

You get a Future object or you can register a callback. You register a callback by implementing the InvocationCallback interface. You specify what you're interested in getting back.

```
class MyCall implements
InvocationCallback<Order>
{
  public void completed(Order order) {
... }
  public void failed(InvocationException
err) {
... }
}
Future<Order> order =
          client.target("http://
shop.com").request().async().get(new
MyCall());
```

The next big feature of JAX-RS is filters and interceptors, which are targeted at framework developers and advanced users. It's not really meant to be used to handle business logic but to add aspects on top of your RESTful Web services, like Gzip encoding or automatically signing a message that you're sending back. Those things are traditionally implemented using filters and interceptors.

Filters and interceptors work in both asynchronous and synchronous environments and the API is use-case driven. Filters are used to modify request and response headers, codes, and methods. For example, an outgoing filter request might change a GET request into a conditional GET request by adding a header. Interceptors deal with the modification of the message body of a response or a request. They wrap around message-body readers and writers. MessageBodyReader/Writer instances are the JAX-RS marshallers and unmarshallers.

The server side has four interfaces: ContainerRequestFilter; ContainerResponseFilter; ReaderInterceptor; and WriterInterceptor.

ContainerRequestFilters modify or preprocess incoming requests on the server side. You can set out filters to be pre or post-resource method matches. Pick what Java method or service you're going to invoke: pre is invoked before the actual work and post is invoked after that match has been made. ContainerRequestFilters can also abort the request and return a response.

Another thing you might want to do with the pre-match filter is content negotiation. Instead of sending an Accept header, you can add a file suffix to your URL like .text.jason.xml. In that case, a pre-match filter would strip the suffix, build an Accept header, and put that into the Java request so that when your Java resource method gets called, the matching will be based on the available information.

For example, here we have a Java header that will contain the HTTP method we want to invoke. The filter is going to see if the request contains that header. If it does, it would take the value of that header and change the Java method that's going to be invoked and applied to the resource method.

```
@Provider
class HttpMethod implements
ContainerRequestFilter
{
  public void
filter(ContainerRequestContext ctx) {
    if (ctx.getHeaders().
containsKey("HTTP-METHOD") {
    ctx.setMethod(
       ctx.getHeaders().getFirst("HTTP_
METHOD"));
    }
} }
```

Post-match filters happen after the resource method is matched. Those are interesting when you want to do things like add behavior by applying an annotation to a particular resource method. For example, you can add authorization support to your RESTful Web services using the RolesAllowed annotation.

```
@Provider
@PostMatch
class Auth implements
ContainerRequestFilter
{
  public void
filter(ContainerRequestContext ctx) {
```

```
    if (isMethodAuthorized(ctx) ==
false) {
       ctx.abortWith(Response.
status(401).build());
    }
}
  boolean isMethodAuthorized(...) {...}
}
```

What about intercepting outgoing responses? This is the job of the ContainerResponseFilter. It's called after the Java resource method is invoked but before marshalling happens. You can use these interfaces to do things like header decoration. It will let you automatically have a cache-control header or generate an ETag or last modified.

Reader/WriterInterceptors are used to wrap the marshalling and unmarshalling of the message bodies of the response or the outgoing request. They're in the same Java call stack as the message-body reader or writer. One example of using a reader or writer interceptor is Gzip encoding.

```
@Provider
class Gzip implements WriterInterceptor
{
  public void aroundWriteTo(
       WriterInterceptorContext ctx) {
    GZipOutputStream gzip =
       new GZipOutputStream(ctx.
getOutputStream());
    ctx.setOutputStream(gzip);
    ctx.proceed(); // call next
interceptor or writer
} }
```

An interceptor can handle automatically enclosing that entity within a multipart/signed document. Maybe you want to post-process a Java object after it's been unmarshalled. Reader/WriterInterceptors allow you to do these things.

On the client side, there are mirrored interfaces. You have a ClientRequestFilter, ClientResponseFilter, and Reader/WriterInterceptors that you can reuse on the client side.

When you're writing different filters or interceptors, you are likely to want them to run in a prioritized order. For instance, you may want your security filters to run first because you don't want to do any processing of HTTP requests until the user is authenticated and authorized. This is where the BindingPriority annotation comes in. You use it to specify a numeric priority. JAX-RS will figure out the set of filters and interceptors it's going to invoke then will sort them based on that value. JAX-RS provides two ways for you to bind a filter interface on a per method basis.

Presentation transcript edited by Mark Little

ABOUT THE SPEAKER

Bill Burke is a fellow and engineer at Red Hat. A long time JBoss veteran, Bill has led multiple different open source projects over the years, the most recent being the RESTEasy project, a JAX-RS implementation. Bill served on multiple expert groups within the JCP including EJB 3.0, Java EE 5, and JAX-RS 1 and 2.0. He is author of O'Reilly's "Restful Java with JAX-RS".