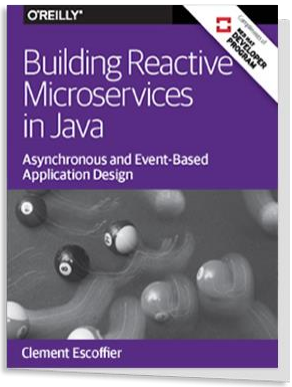
REST Anti-Patterns

By Stefan Tilkov

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

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

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



As with any new approach, it helps to be aware of some common patterns. In the first two articles of this series, I’ve tried to outline some basic ones – such as the concept of collection resources, the mapping of calculation results to resources in their own right, or the use of syndication to model events. A future article will expand on these and

other patterns. For this one, though, I want to focus on anti-patterns – typical examples of attempted RESTful HTTP usage that create problems and show that someone has attempted, but failed, to adopt REST ideas.
Let's start with a quick list of anti-patterns I've managed to come up with:

1. Tunneling everything through GET
2. Tunneling everything through POST
3. Ignoring caching
4. Ignoring response codes
5. Misusing cookies
6. Forgetting hypermedia
7. Ignoring MIME types
8. Breaking self-descriptiveness
   Let's go through each of them in detail.

Tunneling everything through GET

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

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

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

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

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

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

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

Is this a URI that identifies an operation and its parameters, or does it identify a resource? You could argue both cases: This might be a perfectly valid, bookmarkable URI; doing a GET on it might be "safe"; it might respond with different formats according to the Accept header, and support sophisticated caching. In many cases, this will be unintentional. Often, APIs start this way, exposing a "read" interface, but when developers start adding "write" functionality, you find out that the illusion breaks (it's unlikely an update to a customer would occur via a PUT to this URI – the developer would probably create a new one).

Tunneling everything through POST

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

One could argue that tunneling everything through POST shares all of the problems of the GET variant, it's just a little harder to use and cannot explore caching (not even

accidentally), nor support bookmarking. It actually doesn't end up violating any REST principles so much – it simply ignores them.

Ignoring caching

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

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

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

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

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

Ignoring status codes

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

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

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

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

Misusing cookies

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

Cookies are a sure sign that something is not RESTful. Right? No; not necessarily. One of the key ideas of REST is statelessness – not in the sense that a server can not store any data: it’s fine if there is resource state, or client state. It’s session state that is disallowed due to scalability, reliability and coupling reasons. The most typical use of cookies is to store a key that links to some server-side data structure that is kept in

memory. This means that the cookie, which the browser passes along with each request, is used to establish conversational, or session, state.

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

Forgetting hypermedia

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

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

The first indicator of the "Forgetting hypermedia" anti-pattern is the absence of links in representations. There is often a recipe for constructing URIs on the client side, but the client never follows links because the server simply doesn't send any. A slightly better variant uses a mixture of URI construction and link following, where links typically represent relations in the underlying data model. But ideally, a client should have to know a single URI only; everything else – individual URIs, as well as recipes for constructing them e.g. in case of queries – should be communicated via hypermedia, as

links within resource representations. A good example is the Atom Publishing Protocol with its notion of *service documents*, which offer named elements for each collection within the domain that it describes. Finally, the possible state transitions the application can go through should be communicated dynamically, and the client should be able to follow them with as little before-hand knowledge of them as possible. A good example of this is HTML, which contains enough information for the browser to offer a fully dynamic interface to the user.
I considered adding “human readable URIs” as another anti-pattern. I did not, because I like readable and “hackable” URIs as much as anybody. But when someone starts with REST, they often waste endless hours in discussions about the “correct” URI design, but totally forget the hypermedia aspect. So my advice would be to limit the time you spend on finding the perfect URI design (after all, their just strings), and invest some of that energy into finding good places to provide links within your representations.

Ignoring MIME types

HTTP’s notion of content negotiation allows a client to retrieve different representations of resources based on its needs. For example, a resource might have a representation in different formats such as XML, JSON, or YAML, for consumption by consumers implemented in Java, JavaScript, and Ruby respectively. Or there might be a “machine-readable” format such as XML in addition to a PDF or JPEG version for humans. Or it might support both the v1.1 and the v1.2 versions of some custom representation format. In any case, while there may be good reasons for having one representation format only, it’s often an indication of another missed opportunity.

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

Breaking self-descriptiveness

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

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

Summary

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

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