Hypermedia as if you meant it
Dispelling myths about REST

Tomasz Pluskiewicz

PGS Software SA, Wrocław, Poland
t.pluskiewicz@pgs-soft.com

Abstract. This document discusses various myths around Representation State Transfer architectural style and common mistakes made when it is interpreted. The goal of this paper is to combat these mistakes and offer guidance in improving the design of RESTful web services.

Keywords: REST, hypermedia, web apis

1 Representational State Transfer

Representational State Transfer[1] or REST helps build scalable and maintainable web applications or APIs by introducing a number of constraints, which must be satisfied in order for an application to be truly RESTful:

1. Client-server
2. Stateless
3. Cache
4. Layered system
5. Code on demand
6. Uniform interface

As is often the case, each of those constraints comes at a price. Thus not every application has to be really REST-compliant. Yet REST has become such a popular buzzword that everyone wants in and calls their APIs RESTful disregarding the term's actual meaning.

This paper will not go into details about all elements of REST architectural style. For the sake of the argument I will briefly describe the last constraint which causes most trouble to REST adopters.

1.1 Uniform interface

The last constraint is arguably the most important and Roy Fielding calls it a *central feature* of the REST architectural style. Its aim is to simplify the overall architecture by further decoupling clients from servers. It is done by using a standardized rather than application-specific format for communication. The cost of such an approach is possible decrease in efficiency where data has to be transformed. This constraint is defined by four sub-constraints, which realize this decoupling.
Identification of resources. Each resource in a RESTful system is assigned a URI identifier. The identifier is used to access a resource representation. It is important that other than being a pointer to the resource, the identifier does not bear any extra semantics.
Manipulation of resources through representations. The resources are accessed with URIs and the server exposes representations and not resources themselves. This is the biggest difference from RPC-style APIs, where the client operates through a proxy on the actual server-side objects as if they were local such as in CORBA or RMI or WCF.

It is also important that a single resource can be represented in various ways. A resource can be represented as JSON, XML or an image using respective media types.

Lastly, representations are used to capture the indented state of a resource. In RPC the client operates on resources in an object-oriented fashion. In REST however to achieve a similar business operation, the client would change the state of the representation and update the resource by submitting the changed data.
Self-descriptive messages. This constraint should be simple. It means that the resource representations and all meta data attached should be everything the server and clients need to act:

- parsing and semantic rules are defined by the media type,
- caching rules are included as appropriate and well-known headers,
- links to other resources are never hidden in client logic but are made explicit.

Self-descriptiveness also makes it possible for intermediaries/middleware to operate independently of clients, servers and application-specific rules and, most importantly, agnostic to rule changes.
Hypermedia as the engine of application state. The last constraint is the quintessence of REST. Without rich hypermedia the API cannot be truly RESTful, only REST-like or REST-ish.

Hypermedia means data in various formats, all linked by **hyperlinks**. This last constraint is a composition of all the others, because for an application to act upon the resource representation it requires multiple elements, all of which are bound by the REST constraints. Most notably:

1. The media type must be known to the client and rich enough to describe all possible client-server interaction.
2. The client can only follow links included in the representation and cannot construct identifiers without user interaction.

2 Misuse of resource identifiers

There is no such thing a RESTful identifier. Stefan Tilkov discusses[2] various REST mistakes including the misguided notion of RESTful URIs. The misunderstanding of the **identifier** in Representational State Transfer is important and very often repeated. Here is what Roy Fielding writes about identifiers in section 6.2.4 of his dissertation.
At no time whatsoever do the server or client software need to know or understand the meaning of a URI — they merely act as a conduit through which the creator of a resource (a human naming authority) can associate representations with the semantics identified by the URI.

In other words, the identifier is just a pointer used to access resource representations and has no meaning itself. It is a string and all resource semantics are included in the representation. Neither the client nor the server should have to parse and derive any meaning from the identifier. Nor should they have to construct identifiers themselves.

There are many ways the identifier is misunderstood by REST practitioners.

2.1 Hacking URIs

This a very common practice. Of course there is no harm in a hackable URI as long as it is not actually being hacked. There are very good articles on this by Mark Seeman[3] and Ben Morris[4].

The problem arises when clients actually start hacking URIs to construct server requests. Libraries have even appeared to help developers manually build identifiers. It may be easy, but it's definitely not a proper way of handling a REST API. Such practice results in coupling the client with a specific URL scheme. First and foremost it violates the self-descriptiveness of a REST resource, since knowledge from outside the representation is required. What is worse, such design is brittle because the client will break whenever the URL structure changes.

Hackable and readable identifier has only superficial value to the developer and should has no value to the client.

2.2 'Plural or singular'

There are discussions[5] on the web on whether collections URI should be plural, but what about collection members? If there is a collection of books under `/books` resource. I could add a book by executing the `POST` method on the collection resource.

In response the server would return status code `201 Created` and a `Location` header. But what should be the header value? The answer is: **'it doesn't matter'** because the client should follow links rather than mint URLs.

2.3 Hierarchical URIs

There are proponents of URI schemes, where related resources are assigned a hierarchy of a predictable set of identifiers. Actually, the above example is a good start and such design is quite intuitive. This is after all how directory tree of a filesystem works for example. One could expand the bookstore address space to include play acts `http://book.store/books/Hamlet/acts`.

It encourages hacking URIs, which I brought up as first example of URI abuse. Is that good design? Maybe it is easier to assign such identifiers when they are created. But this structure should not matter further on. That is because the client should follow links and not mint URLs.

2.4 Nouns not verbs

Some say that an URL must not contain verbs. Of course identifier such as `http://book.store/books/Hamlet/reserve` may be an indication of bad design, which tries to imitate RPC style API. But that is only true if HTTP verbs are not used correctly. For example `PUT /books/Hamlet/reserve` is a bad idea, because PUT must be idempotent. It should be safe for the client to try again. But replace PUT with POST and it is fine. Also in most cases it is possible to simply change the verb to a noun. To make the identifier more RESTful the URI can be changed to `http://book.store/books/Hamlet/reservation` and used the exact same way. But is the identifier any better? No because it is how the resource is used that determines a good API design, not the URI.

2.5 Content negotiation is not the URI

There are many APIs, which instruct developers to include the format in the URI such as `http://book.store/books/Hamlet.json` or `http://book.store/books/Hamlet.xml`. Do these URIs identify two separate resources? Of course not. They both identify the book *The Tragedy of Hamlet, Prince of Denmark*. One could argue of course that they identify documents[6] about the resource and that is true. But these documents are two completely different resources. And trying to update the book by requesting `PUT /xml/books/Hamlet` would modify the document, if anything. To update the book

itself the actual id `/books/Hamlet` should be used. And to request a specific format there is content negotiation. The server may then redirect to the document or add the `Content-Location` header.

2.6 Query string parameters

I have read that "Query args (everything after the ?) are used on querying/searching resources (exclusively)". But is `/books?id=1234` worse than `/books/1234`? No, because it is the mapping from identifier to resource representation that matters and not whether the URI has a query parameter instead of a segment. If the server assigned it that way, so be it.

Another point is whether a filtered collection resource is another resource? For example, is `/books?author=Shakespeare` a resource? It's probably a matter of semantics and personal taste, but if someone chose to treat is as a separate (derived) resource then clearly the query string parameter is part of the identifier.

3 Links between resources

What really matters is how a URI is used. The worst case scenario is when representations are not linked. REST is defined as an "architectural style for distributed hypermedia systems". The word **distributed** is important, because it means that resources are spread across the web, possibly even across multiple servers or systems. In a REST API these resources are connected by links or hyperlinks, hence the term hypermedia. In practice links must be included in the resource representations, so that clients can follow them to transfer into the next application state.

3.1 Real life links

A good analogy for URI is an address. A physical location of a person's home, work or a doctor's office. Addresses can be quite complicated and contain many parts like street and flat number, floor number, postal code, building name etc. Addresses also vary by country. Let's consider this address in Tokio: 〒100-8994 東京都中央区八重洲一丁目5番3号 東京中央郵便局

The sender does not need to understand its parts. This is why online store can work on an international scale: one can just print this on an envelope and post the letter. All that is important is that it is an address, which links the seller to the buyer.

3.2 REST without links is not REST

Roy Fielding wrote in a blog post:

A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations.

It means that a client cannot rely on a specific URI structure which will break should the server change how it assigns identifiers. Instead of that the client should be coded against a documented set of relations or links between resources.

It may not always be possible to give the client complete links. The web already has a standard solution for such circumstances called URI templates which have been standardized in RFC 6570[7]. There are readily available implementations for many languages.

3.3 Not only inline links

Most examples around the web show links which are included within the response body itself. However, with the HTTP in particular it is not the only option. The protocol defines the `Link` header which can be used to connect resources on the web. It is especially important for media types which do not define a clear link semantics or even any means of including links to other resources. The list includes images, video and to some extent plain JSON, where links are actually indistinguishable from simple text. `Link` header is also useful in media types which do allow linking, but the link does not have a domain-specific meaning for a resource, etc. A common example is collection paging where links to other pages within a larger set are included.

```
HTTP/1.1 200 OK
Link: <http://book.store/books?author=Homer>; rel="first",
<http://book.store/books?author=Homer&page=9>; rel="next",
<http://book.store/books?author=Homer&page=46>; rel="last"
```

Here I use predefined link relations[8]. It is also possible to use custom relations by referring to them with their URIs. This allows link relations to become resources in their own right, with human- and machine-readable representations available. A common example are links to resources, which can help the client build a complete user interface. These can include a representation of common elements such as breadcrumbs, navigation menu or user's authentication status.

```
HTTP/1.1 200 OK
Link: <http://book.store/ui/breadcrubms?for=/books>;
rel="http://book.store/api/breadcrumbs",
<http://book.store/ui/unauthorized-user-menu>;
rel="http://book.store/api/navigation/main-menu"
```

4 Affordances

Affordances are a very important aspect of a REST API because they are precisely what sets it apart from RPC-style APIs. In an RPC style architecture, the client must know up-front what the interface allows. That information can be provided in the form of SOAP's WSDL documents, CORBA's IDLs or exposed via a RMI Registry. It is then used by the client to generate programming objects used to access the remote API as if it was local.

The web browser example is very common among the proponents of REST. The web can work the way it does thanks to the Hypertext Markup Language or HTML. More precisely, it works because disparate parties agreed upon the crucial technical details so that any client understanding HTML could operate in a similar manner. To work in similar fashion a REST API needsa media type rich enough for the server to provide the client with all the information needed to make requests.

Hypermedia controls make it possible to decouple the client from the server thus protecting it from changes in the server's responses and input requirements. How many times have we written client code which checks the state of an object, before calling the server? For example a blogging platform could disallow deleting a post in certain conditions.

Another example is creating resources. It can be implemented as a `POST` to a collection or a `PUT` to an identifier prepared by the server. Or maybe the client should be allowed to construct the identifier from a URI template and `PUT` the representation there? If this information is supplied by the developer it means that the API is violating the hypermedia constraints.

4.1 DIY hypermedia

For an API to be fully RESTful a media type must be used which allows for a rich description of the possible actions. Instead of implementing clients against specific URI structures or conventions, the client should expect named actions. These actions, or relations, are similar to links. Just as the client can expect a specific link it wants to follow, it could also expect an action or operation. The difference is that links are simple URI (or templates), whereas actions require additional details. There are a number of aspects which a media type can describe about an action.

From a pragmatic standpoint, it is possible to just incrementally extend resource representations so as to remove the out-of-band knowledge the client requires.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.warpdrive+json

{
```

```
"actions": [{
"rel": "addToFavourites",
"method": "POST",
"URL": "http://movies.sample/users/tpluskiewicz/favourites",
"parameters": { "movie": "http://movies.sample/movie/10" }
}]
}
```

I've just invented an `application/vnd.warpdrive+json` media type, which includes an array of possible actions and details about them. In this example I'm instructing the client how to add a movie to favourites. Depending on the needs one would extend it with more details:

- media types,
- contract of request/response bodies,
- input constraints (ranges, etc),
- URI templates,
- more.

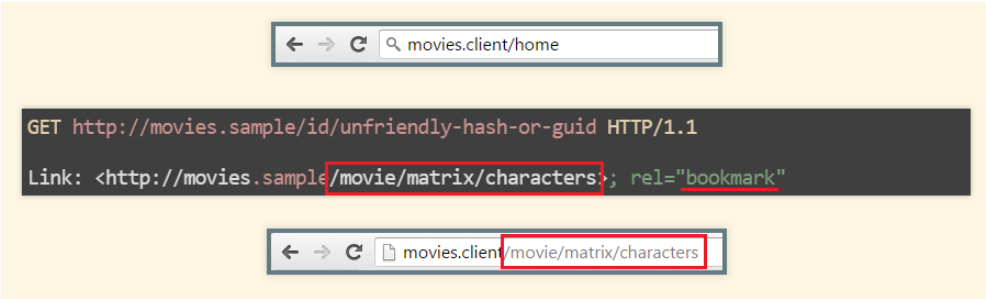
All these details would be defined by the media type itself and implemented in a reusable library. All the API client needs to know is the `addToFavourites` relation name and its meaning.

4.2 Hypermedia types

There are a number of existing media types, which to some extent satisfy the needs of hypermedia constraints: HAL, SIREN, Hydra or NARWHL to name a few. They are not equally powerful, but all have great advantage over a do-it-yourself hypermedia type: they are out there and supported by existing tools. Unless you're creating a dedicated or internal API it is worth adopting any existing media type. Doing otherwise may require a big investment in client libraries, which can interact with the API.

5 Resources As Application State

How do client applications decide what view to present? The most commonly used tool is a *router*, in which developers define URL patterns and their respective views/controllers/etc., which is a terrible idea. Instead I propose reusing the resource identifier whenever possible so that (part of) it becomes the client's address. Or, in the case of user-unfriendly identifiers, a permalink can be added to the representation using a predefined link relation.



Of course this is just a demonstration of a general idea. In a real system the user interface would likely be presenting multiple resource representations and so a complete solution should take that into account.

5.1 Selecting views

The view could be selected based on the resource currently being displayed. A big problem with URL routing is unexpected data. What happens when the server returns some representation which does not *fit* into the view mapped to the given route? A description of an error is one example. Of course, the client will have to take appropriate action and that means more (repeatable) code to maintain.

Instead why not include enough information in the representation for the client to decide what to render? When serving a representations of a book - make it clear in the representation.

```
GET http://movies.sample/movie/10 HTTP/1.1

{
"type": "Movie"
}
```

Here a type relation is added, which the client will use to render the UI accordingly.

It is worth noting that this does not imply *typed resource*, as described by Roy Fielding, because the notion of type does not extend to server's view of resources. It is the type of the representation and the client cannot infer anything about the resource itself based on that information.

6 REST documentation

As with most mistakes around REST, the usual approach to documenting REST APIs is in its core a violation of the resource identification constraint. There are great tools out there like API Blueprint, Swagger or RAML, but they have a significant flaw in common - **they are URI centric**. It is

admirable how most tools take an approach complete with steps for design, testing and API security but in the end the client and server usually rely on URI hierarchies.

The Swagger Petstore example is evident. That kind of API documentation encourages developers to code against rigid URL structures. Also ,most tools do not promote the use of links thus giving little information about possible state transfers.

6.1 Generated clients

Possibly the worst thing about "REST documentation" tools is the idea that they should generate client and server code. There are tools which generate client code from Swagger. All that is required to run the executable, pass in the Swagger URL and the program produces a whole lot of code, which indeed works perfectly with the API, but it has little to nothing to do with REST. In fact such an approach has already been taken by WSDL.

Instead the client should follow links and hypermedia control in a dynamic manner as supplied by the server with resource representations.

7 Versioning and Hypermedia

7.1 Version number in URI

Like most other problems with REST, a common flawed approach to API versioning stems from misunderstanding of the URI. This popular way simply adds a version number as a segment in the resource identifiers. For example identifiers `http://book.store/v1.0/books/978-0321125217` and `http://book.store/v2.0/books/978-0321125217` as identifiers for the blue DDD book by Eric Evans. But do these URLs identify two separate resources? Likely not. The two identifiers are used to access the same resource. This is the most common argument why this approach is wrong. However it is temptingly simple and thus very common.

7.2 Version in Accept header

Another option for serving multiple versions of an API is the use of Accept headers. The client would add the version number to the request. In addition to (or instead of) plain application/json, the server would allow custom media types like `application/vnd.bookstore.v2+json` to serve version 2 of a representation.

It has been noted that for this solution to be symmetrical, the server should not only serve resources with various media types, but should also accept requests with versioned payload. And there are even more edge cases when

`DELETE` method is concerned which does not take neither `Accept` nor `Content-Type` headers.

Also, only part of the server contract could change. For example the server could return updated representation for version 2 but still expect the same request body as in version 1. Does it mean that both need a new version, even if only part of the API changes?

7.3 Version in custom header

The last of the common wrong approaches is leveraging custom HTTP headers for requests, like `X-Version`. The worst that can happen when applying API versioning this way are HTTP intermediaries, which ignore custom headers. Also, both custom header and vendor media type have another caveat. What should the server return when the client does not specify a version? Returning the latest will keep breaking clients while returning the oldest (supported) will likely stall adoption of the improved API.

7.4 What does actually require versioning?

Let's take a step back and think about what aspects of an API can have versions. There are three possible answers: the resource itself (ie. the content), the resource representations or the resource behaviour.
Maintaining versions of a resource. The first possibility is the only circumstance where adding a version number to a resource does indeed make sense. This approach could be used where individual revisions of a resource are served as they change over time. This means that each revision is a resource in its own right and can be interacted with via representations.

Each other path to versioning can usually be mitigated by introducing appropriate hypermedia controls. In the following section I present examples of API evolution, where hypermedia can be used to avoid introducing subsequent API versions.
Handling changes in resource representations. This is where resource custom media types sound like a viable options. A new version of the media type could be introduced to let the clients interact with the new and old representations. For example, in the initial version of an API there can be some sort of a Person resource:

```
{
"name": "Tomasz Pluskiewicz"
}
```

What if the server was to serve `firstName` and `lastName` instead? It is a breaking change at first glance, but there is nothing wrong with including both the old and the new property instead of replacing it.

What about doing a PUT operation on that resource? The media type used should actually inform clients what required inputs are, telling them that in order to modify a Person resource, the name property must be set. After the change the new properties would be used instead and the name property would be marked as read-only and obsolete, which could mean that it will be removed in the future.

Thus if the server provides enough hypermedia descriptions and the client takes advantage of them, **no API versioning is actually required**.
Evolving the resource behaviour. The last case is modifying how the client interacts with the resources. As an example let's consider change in a blogging platform[9]. In the first version the blog is simply posted and published immediately. Such interaction could easily be modelled as a POST to a blog resource.

```
POST http://t-code.pl/blog HTTP/1.1
```

The server may then return a link to the newly created blog post.

```
HTTP/1.1 201 Created
Location: http://t-code.pl/blog/2013/03/REST-misconceptions-part-6
```

So what happens when the contract changes so that a post is not published immediately but saved as a draft? The first POST request could stay the same, so it won't break the client. However, the response would be different:

```
HTTP/1.1 201 Created
Location: http://t-code.pl/drafts/REST-misconceptions-part-6
```

A different Location is returned. The client can now get the draft resource and discover how to publish it. The exact details are debatable and irrelevant. All that matters is that the media type must be expressive enough to describe that interaction in-band as links and hypermedia controls.

Summary. REST has become the de-facto standard way for creating Web APIs. Lack of respect for the architectural style's inherent features causes friction in adoption and repeatable mistakes. The most important element of a REST API are hypermedia controls yet they almost always overlooked. I personally attribute most of the misconceptions and mistakes described in this paper to the omission of the Hypermedia As The Engine Of Application State constraint.

I hope that my example can convince the reader to invest more in this missing constraint. As envisioned by Roy Fielding in his dissertation, I believe that proper inclusion of hypermedia controls can aid the developers in building evolvable servers and robust clients, which will not break upon changes on the server.

With hypermedia the role of the URI diminishes, resource navigation is easier and clients are resistant to changes in interaction flow. Finally less effort is needed to maintain API versions. This leads to potentially less code, further improving the robustness of the application.

Keywords: REST, hypermedia, web apis

Footnotes

1 https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
2 https://www.youtube.com/watch?v=pspy1H6A3FM&feature=youtu.be&t=17m4s
3 http://blog.ploeh.dk/2013/05/01/rest-lesson-learned-avoid-hackable-urls/
4 http://www.ben-morris.com/hackable-uris-may-look-nice-but-they-dont-have-much-to-do-with-rest-and-hateoas/
5 http://stackoverflow.com/q/6845772/1103498
6 https://www.w3.org/TR/cooluris/
7 https://tools.ietf.org/html/rfc6570
8 http://www.iana.org/assignments/link-relations/link-relations.xhtml
9 http://www.troyhunt.com/2014/02/your-api-versioning-is-wrong-which-is.html#comment-2322098802