Representational State Transfer done right

Building Hypermedia APIs

Ruben Deyhle

Stuttgart Media University (HdM)
Nobelstraße 10, 70569 Stuttgart, Germany
rd016@hdm-stuttgart.de
http://hdm-stuttgart.de

Abstract. HTTP is a popular choice for building distributed services and APIs. REST is the architectural style embracing the full potential of HTTP as application protocol. However, many so-called RESTful APIs are not really applying REST. This paper describes the RESTful architecture style with focus on the hypermedia aspect. It debates the use of XHTML as media type for APIs and the resulting opportunity of getting a full-featured, web-based client for free. Finally, it outlines the options for secure authorization on a Hypermedia API.

Keywords: REST, HTTP, Hypermedia, HATEOAS, RESTful, XHTML, API

1 Introduction

The Internet is full of data. Data, that is most likely stored in some database on some server. To utilize this data, in most cases, you simply have a website. Websites are great, because everybody knows how to use them. Every*body* – not *everything*. Websites are for people, not for machines. A web browser can display virtually anything that is supposed to be a website, because HTML is very forgiving. And as long as there is some text, people can read that text and understand it. But for things, for machines, text is just text. They cannot *understand* it. So your data may be greatly accessible for people – but for machines, it is just text they don't understand. They cannot access your data.
Why would machines want to access your data? Because the web is not everything. Sure, a website to use some web service is great and a must-have in most applications. But it is not the ideal way in many cases. A native application is often preferred by users. And this application needs a defined way to access your data. Also, mash-up services, which connect your data to other data to

increase information value, can't do much if you only have a website for human users. For machines, or simply for other applications, you need an API for your data, an Application Programming Interface.

Of course, there is always SQL, the language to talk to your database. But you surely don't want everybody to be able to directly talk to your database, and you may also want to change the way your data is stored in the future. So you need an abstraction layer for third-party applications to access your data. Now you could go about and define your own protocol and just use it to let machines access your data. But this is work, much work. Additionally, you may fail at some firewall, because firewalls usually block most things that are not websites. So it may be a good idea to just use HTTP for your API – nobody says you can only deliver HTML over HTTP.

But how *exactly* do you use HTTP for an API for your web service? A major buzzword for HTTP-based APIs is REST, *Representational State Transfer*. The name does not detail what it actually means in an obvious way, so many people understand many different things when talking about REST. A common understanding is that you should use readable URIs. But that is by far not the only thing that defines a RESTful API, and in fact using readable URIs isn't a requirement to conform to REST standards at all.

This paper elaborates what REST actually is, and how you should design an API to make it great.
It was written for and during the *Ultra Large Scale Systems* course at the Stuttgart Media University (HdM), Germany, in early 2013.

2 Related Work

The term *Representational State Transfer*, or REST, originates from the dissertation of Roy Thomas Fielding [1]. Together with Tim Berners-Lee and some others, he developed HTTP, the web protocol, back at CERN in 1989. Fielding wrote his dissertation "Architectural Styles and the Design of Network-based Software Architectures" in 2000, where he describes and defines REST and its constraints. Basically, he describes a return to basic web technologies to simplify distributed web based systems.

To understand REST, one has to understand HTTP first. The current standard is HTTP 1.1 [2].

In "REST und HTTP" German author Stefan Tilkov gives an elaborated introduction into Hypermedia APIs and an overview in some advanced topics, like security. [6] He also held a Talk at QCon Conference in 2009. [7]
At Twilio Conference 2011, Steve Klabnik held a Talk "Everything You Know About REST Is Wrong" where he describes the different levels "towards RESTfulness" and his relation naming convention.[8]
Jon Moore held a talk about Hypermedia APIs at Øredev Conference 2010, where he demoes his experiences with XHTML as media type for Hypermedia APIs. [9]

3 What REST is not

Simply using HTTP as transport protocol for a web service is not REST. Historically, numerous approaches as to how it could be done right evolved.
The basic idea is to use a standardized notation like XML (*Extensible Markup Language*) to describe data and methods on this data, and sending this XML over HTTP. There are multiple flavors of this, like e.g. XML-RPC (*XML Remote Procedure Call*) or SOAP (originally *Simple Object Access Protocol*). The latter – nowadays not used as an acronym anymore, because it is neither perceived to be simple nor being only for objects – is often understood as the main alternative to REST. XML-RPC, SOAP and similar technologies are often taken together as POX (*Plain Old XML*). They all have in common that HTTP is *only* used as transport protocol. Those services need only a single URI as service endpoint, where everything is sent to and received from. All logic is encapsulated in the messages and has to be pre-defined on server and client.[7, 8]
Examples for this approach are the APIs of Flickr and Google AdSense. Its main disadvantages are the tight coupling of client and server and the lack of visibility. The latter makes caching impossible and may have a bad influence on network performance. The tight coupling makes independent evolution of client and server very hard and results in bad maintainability and decreased extensibility [4]. For most use cases, a RESTful API would be the better choice.

4 How to become RESTful

There is a simple answer: just use HTTP as it was supposed to be used.
HTTP powers the web – the single biggest distributed system there currently

is – and it makes it massively scalable. Its power can also be used for better web-based APIs.
Leonard Richardson describes three Levels one can reach on the way to the "Glory of REST" and out of "The Swamp of POX" ("Plain Old XML"). [5, 8]

4.1 Level 1: Resources and Representations

Instead of having one single, large and complex endpoint for everything, REST uses the divide and conquer principle to break this complexity down into multiple resources.
On a RESTful API, every *thing* is a resource, and has a unique URI. So there is no single endpoint where all communication is tunneled through, instead every resource (or every object in your data model) has a URI. Resources may be static, much like an object that does not change often and is uniquely identifiable. But resources may also be dynamic, for example: the "latest" entry in a collection. Only the semantics should always stay the same for a resource, because the semantics are what distinguishes resources from another. [1, 6]
On requesting a URI, the resource it represents is contained in the HTTP-reply. A common error is to have "typed" resources. A resource is always independent of a specific representation, or media type. This way, a single resource can deliver itself in different media types, depending on what representation the client wants. This Content-negotiation is a part of HTTP, so the requests do not have to be altered to add custom headers. With this, an object (via its URI) can for example be obtained as HTML, as XML, as JSON or even as vCard. The resource always stays the same. Only the form of its *representation* changes. [6]

4.2 Level 2: HTTP Verbs

Another common error is to include methods in resources, like "new" or "update". Instead, a RESTful API has a uniform interface, with a pre-defined, limited number of possible operations. These are, in most cases, the four main HTTP methods: GET, POST, PUT and DELETE. They can be applied to any resource to read or modify it. [2]
The benefit of this is that similar actions can be handled in the same way – and they are well defined by HTTP, including some useful constraints.

GET, to retrieve a resource, is defined as a "safe" action without side-effects that change the data. The client can assume that it is safe to follow a link using GET without modifying or deleting anything on the server. The response is

cacheable; HTTP supports *conditional* GET like "if-modified-since".[2, 6]

POST is defined for adding a new subordinate to a resource, for example inserting a new item into a list. The server then includes the location (the URI of the newly created resource) in its response. POST can also be used for all cases where no other method fits, because it does not guarantee anything – it is not "safe" and cannot be cached. Basically, POST can be used for everything, but does not have any special features either.[2, 6]

PUT is similar, but the request already includes the URI of the enclosed resource – the server should update or create the resource at this specific URI. As opposed to POST, PUT is idempotent, so the request may be sent repeatedly without generating duplicated content; the server makes sure a PUT only creates a resource once. When a POST request does not deliver a reply (due to network issues or something alike), a client does not know if the request was not received or if just the answer was lost. If it is sent again, it is possible that the request is processed twice. When using PUT, the client can be sure that the request is only processed once, regardless of how often it was received.[2, 6]

Finally, DELETE is for deleting a specific resource. Like PUT, it is idempodent – a resource can only be deleted once.[2, 6]

Of course, the server has to guarantee that it handles the HTTP methods as they are defined. It should always respond with the correct HTTP status codes.

So, on this level, instead of packing objects and methods into a single entity, these are split up into multiple resources that can have multiple representations and are accessed through a standard set of methods. [5]

4.3 Level 3: HATEOAS

The key constraint for a RESTful API is HATEOAS: Hypermedia as the Engine of Application State. It is also the constraint that is missing from many self-proclaimed "REST"-APIs.
It means that the resources (as in, the URIs) are not pre-defined. Instead, the server should be free to change them at any time without breaking clients. This is achieved by using Hypermedia, or Hypertext – as the web has been doing it since its beginning. So each representation of a resource should contain the

information where related resources can be found.[1, 6, 8]

This means that a client only has to ”know“ an entry point to the API, most likely the root URI, and is able to discover the available resources from there on: simply by following links. The HATEOAS constraint turns the API into a state-machine, much like an old text-based adventure game: a client is provided with the options where to go next, where to get other resources. If anything changes on the server side, the client will simply get another hyperlink to follow. Furthermore, the API becomes literally self-documenting by implementing HATEOAS.[8, 7]

The linking between resources is done with relations, ideally in a standardized way the media type offers. For example, HTML has the `link` element to reference other resources. They can be distinguished using the `rel` attribute, which contains a relation name. The relation names are the only thing apart from the entry point of the API, that the client needs knowledge of. There are some standard relations like `next` and `prev` defined by the IANA [3], but it can be literally any string.

On this final level, a client only needs to know the root URI of the API, a set of media types or *representations*, and relation names (defining the logic of the underlying data). Client and server are completely de-coupled. The API is highly self-describing, stateless, visible, easily extendable, and has a simple set of methods. It is even very performant due to the document-based messaging (for which the web is optimized for) and network-efficient due to the high visibility of the messages and the resulting high level of cacheability. [5, 4]

5 XHTML as Media Type for Hypermedia APIs

Virtually any data format can be used as media type of an RESTful API. Common choices are XML or JSON. Even simple plain text may be a suitable choice for some resources. However, the HATEOAS constraint prompts usage of a hypermedia-aware media type.
Basic XML and JSON are not hypermedia-aware: there is no “standard” way to define hyperlinks in XML or JSON. Of course, one may define a link relation in e.g. JSON, but it is not a standard. Some XML-based formats, like SVG or Atom, define a link element for hypermedia: so every client understanding those formats already knows where it can find links in such a document. Actually,

Atom is a popular choice for Hypermedia APIs. [7, 8]

But there is one other interesting hypermedia-aware format: XHTML.
Around the turn of the millennium, XHTML 1.0 was the attempt to bring more structure and machine readability to the web and many web developers' format of choice. It was simply an XML-compliant way to write valid HTML, introducing some additional (XML-)constraints to HTML 4, which was the standard at this time.
Today, XHTML fell into oblivion and most web developers use HTML 5 nowadays. But there is also an XML-compliant variant of HTML 5: XHTML 5. It includes all the benefits of HTML 5, like the many new elements and attributes, but is still XML-compliant. So it is easily readable and parseable by a computer, but also knows links, because it is HTML.
Using HTML – or XHTML – as media type for an Hypermedia API has one very simple, but unique advantage: it can be displayed in any web browser and thereby used by humans. It even supports parameterized navigation – using forms.[9]

This way, an API becomes easily usable for both machines and humans, using one and the same media format. For humans, it may be convenient to include styleheets and client-side JavaScript. This doesn't hurt a machine parsing the XML: it will simply ignore the links to these additional resources.
Using XHTML makes it possible for any web service to provide an API that is documenting itself in a convenient way – a client developer may simply look at the source code of the website, as does a client program – and that is a web frontend for the service itself. Utilizing recent web technologies for responsive web design, the one XHTML representation may even become a web app for very different devices.

Of course, it is easy to create a massive overhead in HTML. But it is also easy to outsource additional resources like stylesheets and scripts, and to load additional content just for human users afterwards with JavaScript.
And of course, XHTML is a very non-specific media type. So it is even more important to define and use good, self-describing relations. Steve Klabnik [8] suggests relations with speaking names that are URIs itself, pointing to a description of the relation: e.g. `/rels/newpost` which points to the resource where a new post can be created, and the URI `/rels/newpost` gets documentation

about that fact.

Also, HTML has some limitations that complicate a HTML-based REST API. While it is a great possibility to use HTML forms for constructing follow-up requests, HTML forms are limited to the HTTP methods GET and POST. PUT and DELETE are no valid values for the method attribute of the form element in (X)HTML5. There are two common workarounds for this problem: the desired method can be set in a hidden input element, while still using POST. The server then has to examine the value of that field and handle the POST request accordingly. The other workaround is using JavaScript to handle form submission instead of relying on the browser's capability. This way, actual PUT and DELETE requests can be sent, but the availability of JavaScript is mandatory – which should hardly be an issue in modern browsers.[6]

Overall, XHTML 5 is a great choice as default representation format for a Hypermedia API. It provides a very universal media type for all basic use cases: discovery and documentation of the API in a web browser, XML-compliance for most clients, and even a full-featured web app as reference client.

6 Discussion

The architecture of the World Wide Web, based on the HTTP protocol, is a great base to build APIs. It is heavily optimized for delivery and modification of resources as documents – which is all an API needs. Using it is not only simple, but also brings all the benefits of HTTP – like caching and the default set of methods – for free.
But there may still be some open questions regarding security and authorization in REST.

6.1 Security and Authorization

The messages in a Hypermedia API are inherently plain text. To provide end-to-end encryption, HTTP banks on transport-based security by using SSL. This is also the way to go with an Hypermedia API, or REST API. HTTPS is an established standard used by many websites and should be used whenever sensitive data is transmitted. Of course, a valid SSL certificate is crucial for real security. But this is a solved problem.[6]

HTTP also solves authorization. It provides two authorization mechanisms, basic and digest authentication.
HTTP-Basic-Auth includes username and password in the HTTP request, both in cleartext (base64-encoded). This is the most simple form of authorization and widely supported. Of course, it should only be used with HTTPS to provide encryption. HTTP-Digest-Authentication solves the problems of Basic-Auth by encrypting the password and introducing a timestamp, but is only insufficiently implemented in many clients and thus not very commonly used. Also, usage of SSL makes Digest-Authentication superfluous, Basic-Auth is just as good. When Basic-Auth is used, web browsers display a modal window prompting for username and password. And Basic-Auth in conjunction with SSL is also the best choice for REST APIs, because it is simply HTTP(S).[6]
However, most websites don't use Basic-Auth and instead implement a proprietary cookie-based login mechanism, because Basic-Auth has some disadvantages. First, there is no such thing as a logout. Once logged in, the browser continues sending the login information on every request. Second, the login window is not customizable by a website, so it is not possible to show links for registration or password reset. And finally, on many websites login may be optional; in this case, a visitor would never see the login window.[6]
Because of these reasons, a proprietary cookie-based login is necessary for most cases. However, this is not standardized and does not correspond to the REST principles. Tilkov [6] suggests providing both authorization methods when using HTML, Basic-Auth for clients and cookie-based login for web site visitors. Another option is the usage of a token-based system like OAuth. A client then sends his auth token along with the request in the HTTP header, which also blends greatly with REST.[6]

6.2 Conclusion

HTTP is a great choice for building distributed services. While SOAP and similar approaches disregard the principles of the web, REST embraces it and enables the full power of HTTP for an API.

A RESTful API is defined by four constraints: the use of resources, identified by URIs; representations of these resources, ideally in an hypermedia-aware format; a fixed set of verbs as uniform interface on all resources, in most cases the four basic HTTP verbs; and finally hypermedia as the engine of application state, linking the resources together and making them discoverable using the

hypermedia principle.

XHTML is an interesting choice as representation media type because it is XML-compliant and therefore easily machine-readable, and it can be parsed in any web browser, making the API accessible for humans as well.

References

1. Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Dissertation. University of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
2. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. 2009. http://www.ietf.org/rfc/rfc2616.txt
3. Link Relations. Internet Assigned Numbers Authority (IANA).http://www.iana.org/assignments/link-relations/link-relations.xml
4. Algermissen, Jan. Classification of HTTP-based APIs. Feb 25, 2010. http://nordsc.com/ext/classification_of_http_based_apis.html (retrieved Feb 8, 2013)
5. Fowler, Martin. Richardson Maturity Model. Mar 18, 2010. http://martinfowler.com/articles/richardsonMaturityModel.html (retrieved Feb 8, 2013)
6. Tilkov, Stefan. REST und HTTP. Dpunkt Verlag 2011. ISBN 978-3898647328
7. Tilkov, Stefan. REST: A Pragmatic Introduction to the Web's Architecture. QCon London. Jan 29, 2009. http://www.infoq.com/presentations/qcon-tilkov-rest-intro
8. Klabnik, Steve. Everything You Know About REST Is Wrong. Twilio Conference 2011 San Francisco. http://vimeo.com/30764565
9. Moore, Jon. Hypermedia APIs. Øredev Conference Malmö. Nov 10, 2010. http://vimeo.com/20781278