FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

WEB APIS: FROM START TO FINISH

eMag Issue 22 - January 2015

InfoQ

THIS ISSUE
A BUSINESS PERSPECTIVE ON APIS

INTERVIEW
An Interview with HAL Creator Mike Kelly

INTERVIEW
Roy Fielding on Versioning, Hypermedia, and REST

A Business Perspective on APIs

This article examines APIs from a business perspective, whether or not they are open and overtly monetized. It covers the importance of tying your APIs back to your business value, looks at the type of data that should be used, and studies the success stories of Amazon and Twilio.

A Web API Design Methodology

This article provides a brief overview of the design methodology covered in the book "RESTful Web APIs" by Richardson and Amundsen.

An Interview with HAL Creator Mike Kelly

As part of our ongoing series on Web APIs Mike Amundsen talks to Mike Kelly about his reasons for creating HAL and his experiences over the last three years with web developers and the API community.

WEB APIS:
FROM START TO FINISH
InfoQ
THIS ISSUE
A BUSINESS PERSPECTIVE ON APIS
INTERVIEW
An Interview with HAL Creator Mike Kelly
INTERVIEW
Roy Fielding on Versioning, Hypermedia, and REST

Implementing Hypermedia

In this article, we'll talk about four different real-world implementations of hypermedia: how you may already be using hypermedia through image links, how GitHub uses the Link header for pagination, using hypermedia in constrained systems like iOS, and how Balanced uses hypermedia principles to build their product.

REST-y Reader

Rounding out our first Web APIs series Mike shares books je recommends for those who want to learn more about designing, impelementing, and maintaining APIs for the Web.

Roy Fielding on Versioning, Hypermedia, and REST

Roy Fielding talks to Mike Amundsen about versioning on the Web, why hypermedia is a requirement in his REST style, the process of designing network software that can adapt over time, and the challenge of thinking at the scale of decades.

MIKE AMUDSEN In his role of Director of Architecture for the API Academy, Amundsen heads up the API Architecture and Design Practice in North America. He 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" published in 2013. Mike's 2011 book, "Building Hypermedia APIs with HTML5 and Node", is an oft-cited reference on building adaptable Web applications.

A LETTER FROM THE EDITOR

Designing, implementing, and maintaining APIs for the Web is more than a challenge; for many companies, it is an imperative. This eMag contains a collection of articles and interviews from late 2014 with some of the leading practitioners and theorists in the Web API field. The material here takes the reader on a journey from determining the business case for APIs to a design methodology, meeting implementation challenges, and taking the long view on maintaining public APIs on the Web over time.

In the first article, Matt McLarty examines APIs from a business perspective. He covers the importance of tying APIs to your business value and studies the success stories of Amazon and Twilio. Matt is Vice President of the API Academy from CA Technologies where API design, implementation, and alignment is a primary focus. Anyone charged with aligning APIs to business goals will find valuable lessons in his piece.

I also had the chance to sit down with Mike Kelly to talk about his Hypertext Application Language (HAL) format, currently the most popular of the new "hypermedia" message designs in use today. Mike Kelly is a well-known voice in the API and REST communities and his observations on why HAL has gained traction with companies like Amazon and what this means for the future of "Hypermedia APIs" is enlightening.

As the runtime interface for Web APIs moves toward one that supports backward-compatible changes and allows for some level of evolvability, a consistent design methodology that makes it possible for both service providers and consumers to work indepedently on the same interface — even when separated by time and distance — becomes critically important. To that end, my contribution to this series is a brief overview of the Web API design methodology covered in the book "RESTful Web APIs" which I co-wrote with REST API guru Leonard Richardson.

Business strategies and design methodologies are not worth much without actual implementation guidance and Steve Klabnik's piece on four real-world examples of hypermedia API usage adds a great deal to the series. He points out that you may already be using hypermedia through image links, how GitHub uses the Link header for pagination, the advantage of using hypermedia in constrained systems like iOS, and how a Web payment startup, Balanced, used hypermedia principles to build their product from the ground up. Steve is a Rails committer, Rust contributor, author of "Rails 4 in Action", "Designing Hypermedia APIs", & "Rust for Rubyists" and he's a recognized implementation authority in the Web community.

And no eMag on Web APIs would be complete without a chance to hear from person who coined the term "REST" — Roy Fielding. Roy is a Senior Principal Scientist at Adobe and a major force in the world of networked software. His name appears on more than a dozen specifications with both the IETF and W3C and, through his various roles in the Apache Software Foundation, he helped shape the world of open source software. I had the honor of interviewing Roy for this series where he talked about versioning on the Web, why hypermedia is a requirement in his REST style, the process of designing network software that can adapt over time, and the challenge of thinking at the scale of decades. His view of the "state of Web APIs" twenty years on is both interesting and challenging.

Rounding out the set, I share a list of books and articles I highly recommend for those who want to learn more about designing, implementing, and maintaining APIs for the Web. The set is not limited to books on HTTP, the Web, and APIs specifically, though. I also include a handful of references to Information Theory, complexity, and usability design — all topics which come into play when creating robust, long-lasting Web APIs.

I really enjoyed putting this collection together and was inspired by the advice and observations of the series contributors. I hope you find it valuable and that you, too, will find the content enlightening and helpful as you work to create great APIs for the Web.

AP

READ ONLINE ON InfoQ

MATT MCLARTY (@mattmclartybc) is vice president of the API Academy from CA Technologies. The API Academy helps companies thrive in the digital economy by providing expert guidance on strategy, architecture, and design for APIs.

A BUSINESS PERSPECTIVE ON APIS

APIs are at the heart of every major information technology trend. Mobile devices, cloud computing, the Internet of Things (IoT), big data, and social networks all rely on Web-based interfaces to connect their distributed components and deliver innovative and disruptive solutions to every industry in global business. Smart grid technology is transforming the energy industry. Connected-car solutions are viewed as a key differentiator in the automotive industry. Amazon is disrupting every industry it touches. In all of these cases, APIs are both catalysts and enablers.

APIs are at the heart of every major information technology trend. Mobile devices, cloud computing, the Internet of Things (IoT), big data, and social networks all rely on Web-based interfaces to connect their distributed components and deliver innovative and disruptive solutions to every industry in global business. Smart grid technology is transforming the energy industry. Connected-car solutions are viewed as a key differentiator in the automotive industry. Amazon is disrupting every industry it touches. In all of these cases, APIs are both catalysts and enablers.

Because of this business impact, much has been written about "the business of APIs". On the open Web, there is a distinct business model for using APIs as an external channel for innovation and revenue. This is documented comprehensively by Kin Lane on his API Evangelist site, and summarized elegantly by Mehdi Medjaoui in this recent post. However, this open-API model represents the tip of the iceberg when it comes to API adoption across the technology spectrum. In fact, the majority of Web APIs are hidden within the solutions they enable. In this sense, the business of APIs is just business itself.

This article will examine APIs comprehensively from a business perspective, whether or not they are open and overtly monetized. I will cover the importance of tying your APIs back to your business value, look at the type of data that should be used, and study the API success stories of Amazon and Twilio. I hope that these lessons will help you build useful and usable APIs.

Measuring API Business Value

The general business value of APIs can be measured. It all starts with data. For many companies and organizations, the data they collect is viewed as a liability. Servers and storage are expensive. However, in the increasingly digitized world, data can also be a precious asset. Data provides insights on clients that can be turned into business opportunities and new revenue streams. The big-data craze seeks to sort out the digital confusion through high-value analytics. The imminent IoT explosion will bring an exponential increase in data, making it even more critical for companies to be on the right side of the data ledger.

The degree to which a company's data is an asset as opposed to a liability is driven by three things: its accessibility, accuracy, and applicability.

Any web API provides accessibility to some data to some degree. Valuable APIs provide access to accurate data that applies to their core business. This enables companies to achieve an iterative approach I call "Data-Enabled Disruption", which I will explain below. Furthermore, these three data attributes provide a methodology for determining which data and services should be exposed through APIs, and how those APIs are implemented (see table).

Data Applicability	Will this data help drive my key business goals? Does this data differentiate my business? Will I commoditize this data if I expose it externally?
Data Accuracy	How current is the data being provided? Is the data coming from an authoritative source? Is data being accessed by the right end users for the right purpose?
Data Accessibility	Which data is available for programmatic consumption? What are the different ways this data can be accessed? How easy is it for developers to build apps that use this data? Can the data access scale to meet the client needs?

Examining this methodology from the perspective of APIs, these three data attributes can be consolidated into two attributes of APIs:

- "Useful APIs" provide accurate and applicable data.
- "Usable APIs" provide accessible data.

Obviously, the most valuable APIs are both useful and usable. However, in order to define these API attributes further, let's examine each one individually.

Useful APIs

A common mistake people make when developing APIs is to assume that all data is useful. There is a widely propagated myth that if you open up your data, magical developers will appear and spread pixie dust on it that increases your revenue, creates innovative ideas, and opens up new business channels. APIs and open data are not enough on their own. This "medium is the message" thinking was responsible for many of the failed SOA initiatives that occurred in enterprise integration over the last decade. I recall one very large enterprise that was embarking on a $50M+ SOA and private cloud initiative. They were baffled when I asked what services they would be exposing to which clients. All they cared about was building the infrastructure. Needless to say, the initiative failed.

The good news is that the desired revenue and innovation can be achieved if the right data is exposed through the right APIs. Google Maps leveraged Google's dominant Web presence to deliver an API-based service that filled a market gap. The service was so useful that Google was able to monetize it at a premium. Through its API, Google Maps was embedded in the original iOS platform, and the initial poor reception for Apple's replacement only emphasized its value. Social networks, led by Facebook and Twitter, are a medium whose growth and success are inexorably linked to their use of APIs in order to facilitate Web linking and mobile adoption. Useful APIs have even impacted federal election campaigns.

The Amazon.com API story

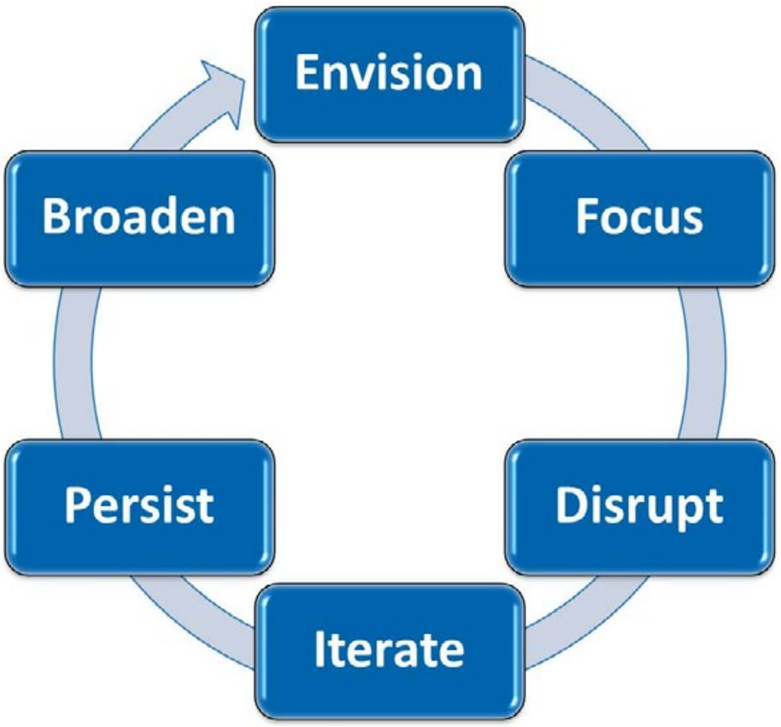
The potential for business success through APIs runs much deeper inside organizations than is apparent from the world of open APIs. This potential was heavily exploited at Amazon.com, a company whose API origins were internal. Amazon's enduring API-enabled success is arguably unmatched in any industry. The company provides a number of lessons through its use of APIs that can help other organizations achieve business success with their own API programs.

The first and most obvious lesson from Amazon is how to position APIs as building blocks for product and solution offerings. Brad Stone spends a chapter in his book on how fundamental APIs are to the technological landscape at Amazon. Kin Lane summarizes nicely how adamant Jeff Bezos was and is in ensuring interfaces at Amazon provide programmable access. According to these reports and others, product managers must identify the lowest common denominators of business value in their offering proposals. The technical teams then use these "primitives" to create APIs that expose this business value to other developers who create the remainder of the solution. The logic behind this is that these increments of business value can be used and combined easily for future offering development.

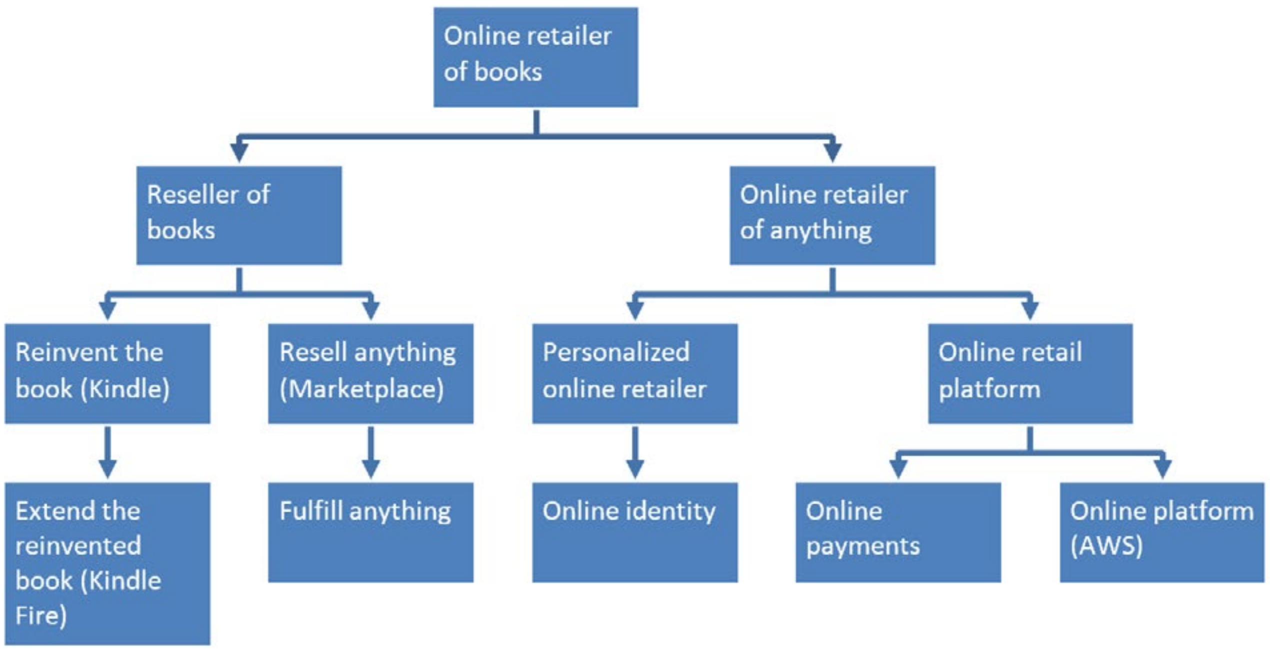
This is how Amazon Web Services (AWS) could be delivered and enhanced so rapidly: Amazon had API-enabled its infrastructure services in order to optimize the process of expanding its capacity. AWS came about by turning those internal functions into an external product. As you can see, Amazon goes a step further than just ensuring the APIs it provides include useful data. It inverts this principle and instead ensure that every quantum of data employed in a solution sits behind an API.

The second API lesson from Amazon is how to use an API-based approach to collect, analyze, improve, and distribute valuable data. Jeff Bezos had an epiphany about Amazon's core value proposition in the early days of online bookselling: "We don't make money when we sell things. We make money when we help customers make purchase decisions." He aligned the execution of the company with this insight, and this led to strategic moves such as personalization and the broadening of channels.

In fact, it is clear from the above-stated value proposition that data—applicable, accurate and accessible data to guide customer decisions—was more vital to Amazon's success than books or any other goods that they might sell. At Amazon, APIs are the conduit for the continual accumulation and improvement of data. This cyclical process fuelled Amazon's growth. I call this 360 degree cycle "Data-Enabled Disruption (DED)", and summarize it as follows:



▶


Online retailer of books
Reseller of books
Online retailer of anything
Reinvent the book (Kindle)
Resell anything (Marketplace)
Personalized online retailer
Online retail platform
Extend the reinvented book (Kindle Fire)
Fulfill anything
Online identity
Online payments
Online platform (AWS)

As a result of DED, many of Amazon's former competitors are now dead. By utilizing APIs at all points of the data lifecycle, Amazon is able to continuously improve the accuracy, applicability, and accessibility of its data.

The final API lesson from Amazon derives from the disciplined way the company is able to balance its tactical delivery with its strategic positioning. This was evident from the start, when Jeff Bezos recognized the potential of the World Wide Web and had a vision to create "the everything store". Knowing that he needed to start small, Bezos analyzed the market and identified online book-retailing as a ripe entry point with an optimized supply chain. This paradoxical approach of ensuring timely execution while maintaining religious devotion to the future vision is now a tenet of the company culture at Amazon, where solutions must add value in both what they deliver and what they enable. An API-based delivery approach supports this principle, since APIs power new applications and services while facilitating future use cases. This iterative methodology has fuelled the company's relentless expansion ever since.

Each of these services has an associated external API, but each one is also built on the API scaffolding provided by earlier offerings. As companies look to grow their businesses vertically and horizontally, they can utilize the Amazon approach that is predicated on useful APIs: continually collect and harvest applicable data, use APIs as the common access points to that data, deliver only what is needed in the short term with the long term firmly in mind, then expand from a position of strength.

Usable APIs and the importance of API design

As remarkable as Amazon's success has been and as fundamental as its APIs have been to that success, their external APIs are not generally regarded as the best designed and easy to use. Still, with the explosive growth of APIs and the increasing acceptance of their necessity, API usability is a key differentiator for companies that want to dominate their industries or even startups that just want to establish their innovative service.

Mobile devices and the consumerization of IT are creating a generational paradigm shift in enterprise application development. Previously, there has been dumb terminal-mainframe, then client-server, and most recently the distributed topology of the Web. I have talked previously about what I call "the shedding of tiers", moving away from the n-tiered Web model to an API-centered topology in support of mobile and cloud. This shift includes a move away from Java Enterprise Edition to JavaScript and its derivatives as the languages of choice. All of this means that there is a new wave of developers who will increasingly be the ones building new enterprise solutions. These developers are conditioned to seek out APIs for functions that they require. Companies can anticipate this shift and cater to this new generation of developers by emphasizing the usability of their APIs.

Consider the telecommunications industry. For years, the big carriers have competed vengefully against each other while trying to deliver value-added services that go beyond network delivery. This industry has experienced incredible disruption over the last 15 years, with the advent of VoIP, the convergence of business and operator services, and the revolution in mobile-device services. In each of these disruptions, APIs played a role. In spite of their dominant position in traditional telco services, the big carriers struggled to take advantage. They had even more difficulty when they tried to collaborate on initiatives like Parlay X and OneAPI, as summarized in this article by Alan Quayle. So, if the big guys weren't exploiting these opportunities, who was?

Twilio was founded in 2007 with the goal of providing easy-to-use voice and text contact services, all deployed in the cloud. It demonstrated a platform mentality from the outset and recognized that its API would be its number-one business channel. SMS and VoIP services are certainly useful, but in order to compete with the big carriers, Twilio needed to deliver more than a set of commoditized telco services.

Twilio's key insight was to recognize that the initial clients of its service was not the end user of the app calling the API but the developers creating the apps themselves. It knew that the greatest growth lay in the mobile app-development segment so it came up with a set of metrics to measure how well it was serving this client group. In addition to measuring traditional end-user statistics like end-to-end API-call response times, it measured the time it would take new developers to register for their APIs, and set aggressive targets. This focus improved usability and made Twilio clearly different from the big telcos. When it came time for app developers to choose a SMS or VoIP provider for their apps, this rapid, carrier-agnostic service stood well above its competitors. By providing a useful API, Twilio could justify charging for this service and make money on every API call. By providing a usable API, it drove up these volumes and along with it its revenues.

Industries beyond telecommunications are ripe for data-enabled disruption too. Consider Ingenie, the insurance startup that found an opportunity in the way actuarial-based pricing punishes the 16 to 25-year-old demographic. The company collects individual driver data through a proprietary smart device in the car, then use that to offer insurance discounts to this group. Useful and usable APIs allowed Ingenie to use data-enabled disruption and "Twilio" the insurance industry.

Useful and usable API lessons

To recap, there are a number of steps you can take to ensure the success of your API program:

- Align your APIs with your business strategy.
- Do this by including accessible, accurate and applicable data in your APIs.
- Make sure your APIs are both useful and usable.
- Like Amazon, establish a disciplined culture of iterative data-enabled disruption.
- Like Twilio, create a phenomenal API developer experience to differentiate your business versus larger competitors.

Follow these lessons, and you will set a course for business success that is bolstered by your APIs. You are the best judge of what will make your APIs useful to your clients. Make sure to read the other articles in this series, and you will learn some great tips on how to make those APIs usable. ■

A common mistake people make when developing APIs is to assume that all data is useful.

- Matt McLarty

An Interview with HAL Creator Mike Kelly

by Mike Amundsen

Mike Kelly is a software entrepreneur and founder of the API consultancy Stateless - assisting companies with their strate gy, design and implementation challenges. He lives just outside London in a small city called Winchester with my wife and three children. Mike work primarily with clients in London and Europe and recently traveled to Detroit, Michigan as a guest speaker and panelist for Apigee's API Craft event where he talked about Hypermedia APIs and his very popular hypermedia design named Hypertext Application Language (HAL).

In 2011, he released the Hypertext Application Language (HAL) media type for APIs. Since that time, HAL has grown to be the dominant hypermedia format for APIs.

Kelly does not often attend public events but recently made an exception and participated in the 2014 API-Craft event in Detroit, where he talked about HAL and about the future of hypermedia and APIs in general.

Not long after that event, Mike Amundsen interviewed Kelly to talk about his reasons for creating HAL and his experiences over the last three years with Web developers and the API community.

In July 2011, you registered the HAL media type with the IANA. What was your motivation to create HAL?

I was working with a client on the new API of their SaaS product. The roadmap for the product included significant back-end changes that were going to affect URL structures in the API. I wanted to design the API so that we could roll out those changes with as little friction as possible, and hypermedia seemed like the ideal style for that.

So, one the key goals was to lessen the impact of change over time?

Right. Another objective was to optimize the developer onboarding experience since API integration was a key value proposition of the product and a key driver of sales. For me that meant we needed to have clear, accessible documentation and the API itself had to be easy to play around with.

For that, I wanted to build an API browser that would allow developers to jump around and explore the API and its documentation in a similar way you do with a Web app. So I needed a general-purpose format we could use across the whole API that we could then build a browser against.

So you initially created HAL to solve a problem for one of your projects. To use Eric Raymond's words, you "scratched an itch" you had. If HAL was built to solve your SaaS problem, how did it grow to be such a common format used by so many others?

The approach seemed to work really well, so I shared the idea with some other people in the API space by sketching out some examples of a new general-purpose format based on JSON. Feedback was really positive and a couple of people even started to adopt this sketched-out format in their APIs. Momentum built and, to cut a long story short, I ended up having to build a Web page describing the sketch in more detail. I called it the "Hypermedia Application Language" and this eventually evolved into a formal Internet draft.

HAL was the first in a wave of new API-centric media types registered in the last few years. What do you think is going on here? Why are these new designs popping up now?

I think there's a growing realization that although our APIs may cover very different business domains, there's a lot of common territory shared in terms of architectural style. Media types are a way we can start to collectively map some of this territory on the web of APIs and allow us to develop and share tools and techniques that are applicable to a broad range of problems.

One of the important design choices in HAL is that it focuses on links and resources in the message and relies on human-readable documentation to explain how to manipulate those resources using HTTP methods. Why did you design HAL this way?

I find that most APIs I work on need to optimize the developer onboarding experience because it is what drives up key success metrics like API-user acquisition and activation. In my opinion, to achieve this you need concise API messages and rich, human-readable documentation that clearly demonstrates to developers what those messages will look like.

So your design relies on both a machine-readable format and human-readable documentation. What about the idea that the message should contain all the details for executing a transition such as the arguments, method names, etc.?

I think there is a law of diminishing returns when adding features to a media type. More is not always better since more features mean more of a burden for consumers of the format. HAL is my interpretation of where the sweet spot lies in hypermedia-type design.

In early 2014, Amazon.com announced that its AppStream API was using HAL. What role did you play in helping Amazon learn how to use HAL for their API?

Encouragingly, I didn't play any direct role at all. I think this is a good indication that HAL's design and specification are heading in the right direction. APIs using HAL are popping up all over the place. It's impossible to keep track of them!

So the API community is picking this up on their own and solving problems with HAL. Uou have a mailing list for HAL (HAL Discuss) and it seems pretty active with many people asking questions and sharing experiences. I noticed, though, that you don't seem to post a lot there. Why is that?

Another promising thing that's been happening is that I do not have to answer many questions or give advice on the mailing list any more, since the community of HAL adopters are actively sharing their experiences and interpretations of the spec. I think that's a really healthy thing because, at this point, HAL has evolved beyond my intended design and is actually about what I managed to communicate and how that has been interpreted and deployed by the community. Fortunately, those two things seem to be largely in unison!

I don't have any figures, but it seems that among the new round of designs that have come out in the last few years, HAL is the most-used hypermedia format. Is that true? If so, why do you think that's happening? Why are so many developers attracted to HAL?

I don't have any figures on that either, but I would agree it does seem that way. I'd like to think this was because HAL's design is well balanced and introduces hypermedia without compromising too much on the simplicity of JSON. In reality, it's ►

probably partly that and partly that it just appeared in the right place at the right time.

This gets back to your design decisions. For example, one topic that comes up with some regularity on the HAL Discuss mailing list is the fact that HAL lacks inline forms like those in HTML, Siren, Collection+JSON, and a few others. What do you say to people who want to add these details to the HAL spec?

Whilst it is possible to try to add more dynamic affordances to HAL, I've actively resisted doing that because I feel that the added complexity of doing so damages the developer experience by making the messages less concise and at the same doesn't solve enough real-world problems that cannot be solved using a simpler design.
One of the reasons often cited for needing forms in an API message is that they're required for a GUI application. My issue with this is that HTML is a perfectly adequate hypermedia format for representing GUI forms - it's ubiquitous and developers are very familiar with it. Most of the media types that are introducing forms in JSON seem to be reinventing that relatively complicated wheel, which seems like a lot of effort for not very much gain.

So you really see HAL as a format for machine-to-machine use? And you don't see this adding this information inline as very useful for machines?

My opinions on forms intended for machine-to-machine are probably a whole other interview on its own! In summary, I'm very skeptical and I haven't seen many compelling examples (theoretical, or real world) that convince me they are worth the added complexity.
Having said all of that, a while ago, I did sketch out an extension to HAL called HALO which would introduce this type of dynamic affordance. People are interested in it but, to be honest, I don't really feel convinced enough about its usefulness to push it forward right now.

Over a year ago, you started a HAL RFC. The current document has expired. Do you have plans to pick this up again?

Yes. I recently sent out a call to action on HAL Discuss asking for some final feedback and adjustments to the specification. If any readers have input to share here, please raise a pull request!

Looking back, is there anything you might have done differently with HAL? Anything you know now that you wish you'd known then?

I'm fairly happy with the way things have gone and how things have worked out.
I probably should have written and released the HAL browser sooner, since that has really made things click for a lot of people. Also, I should have proactively engaged high-profile open-source Web frameworks like Rails::API and Ember.js in their early stages since they're good candidates to adopt a general-purpose media type like HAL as their default message format, which would have made adopting HAL even easier. I should mention there are actually third-party HAL libraries for both Rails and Ember, and for many other languages and frameworks.

It seems that every few months, a new format appears. As one of the first in this new wave of media-type designers, do you have any advice about creating a new media type?

My advice would be not to worry too much about having the longest feature list. The shorter your specification, the better. Scratch an itch; don't look for a problem to solve. Don't try to sound clever; use plain language and simple examples. Make sure you share it with others as early as possible so that you can iterate on the wording and make it as clear as possible. And finally, try to avoid reinventing HTML in JSON! ■

A Web API Design Methodology

Mike Amundsen in his role of Director of Architecture for the API Academy, Amundsen heads up the API Architecture and Design Practice in North America. He 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" published in 2013. Mike's 2011 book, "Building Hypermedia APIs with HTML5 and Node", is an oft-cited reference on building adaptable Web applications.

Designing Web APIs is more than just URLs, HTTP status codes, headers, and payloads. The process of design --what is essentially a "look and feel" for your API --is very important and is well worth the effort. This article briefly outlines a methodology that results in an API design that takes advantage of both HTTP and the Web. And it can work for more than just HTTP. If, at some point, you need to implement the same service over WebSockets, XMPP, MQTT, etc., most of the features of this design will work the same. That can make supporting multiple protocols in the future easier to implement and maintain.

Good design goes beyond URLs, status codes, headers, and payloads

Typically, Web API design guidance focuses on the common features such as URL design, proper use of HTTP features such as status codes, methods, headers, and the design of payloads that hold serialized objects or object graphs. These are valuable implementation details but not much in the way of API design. And it is the design of the API --the way the essential features of the service are expressed and described --that can make an important contribution to the success and usability of your Web API.

A good design process or methodology defines a consistent, repeatable set of steps to employ when working to expose a server-side service component as an accessible, usable Web API. That means that a clear methodology can be shared with developers, ▶

designers, and software architects in order to help coordinate activities throughout the implementation cycle. An established methodology can also be refined over time as each team discovers ways to improve and streamline the process without adversely affecting implementation details. In fact, implementation details can change (e.g. which platform, OS, frameworks, and UI style to employ) independently of the design process when these two are cleanly separated and defined.

A seven-step API design methodology

What follows is a brief overview of the design methodology covered in the book *RESTful Web APIs* by Richardson and Amundsen. There is not enough room here to go into depth for each step in the process but this article can give the big picture. Also, the reader can use this overview as a guide for developing a unique Web API design process that fits the local skills and goals of your group.

Seven steps seem like quite a few. In reality, there are only five design steps and two additional items in the list, including implementing and publishing. These last two round out the process to provide an end-to-end experience.

You should plan to reiterate through these steps as needed. You may draw state diagrams and get through Step 2 then realize there are more parts to be listed in Step 1. When you get around to writing the code (Step 6), you may discover a number of things missed in creating the semantic profile (Step 5), etc. The key is to use the process to expose as many details as possible and to be willing to go back a step or two in order to capture the items you missed along the way.

Iteration is the key to building a more complete picture of your service and clarifying how it can be exposed to client applications.

Step 1: List all the parts

The first step is to list all the pieces of data a client application might want to get out of the service or put into it. We'll call these the semantic descriptors. "Semantic" because they deal with the meaning of data in the application and "descriptors" because they describe what is happening in the application itself. Note that the point of view here is that of the client, not the service. It's important to design the API as something the client will be using.

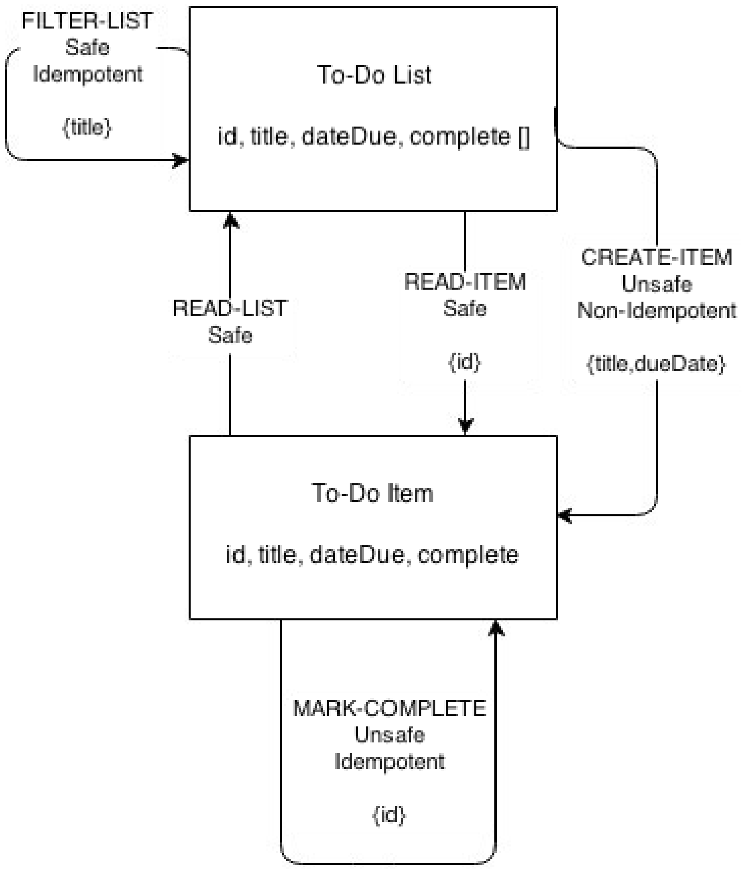
For example, in a simple app like a to-do list, you might find the following semantic descriptors:

- `id`: the unique identifier for each record in the system
- `title`: the title of each to-do item
- `dateDue`: the date the to-do item is due for completion
- `complete`: a yes/no flag indicating whether the to-do item has been completed

In a full-featured application, there could be many more semantic descriptors to cover things like categories of to-do items (work, family, gardening, etc.), user information (for a multi-tenant implementation), and so on. We'll keep this one simple in order to focus on the process itself.

Step 2: Draw state diagrams

The next step is to draw state diagrams for the proposed API. Each box in the diagram represents a possible representation --a document that includes one or more of the semantic descriptors identified in Step 1. You can use arrows to indicate transitions from one box to the next – from one state to the next. These transitions are initiated by protocol requests.



Don't yet worry about indicating which protocol method is used in each transition. Just indicate whether the transition is safe (e.g. HTTP GET), unsafe/non-idempotent (e.g. HTTP POST), or unsafe/idempotent (PUT).

(Idempotent actions are repeatable without unexpected side effects. For example, HTTP PUT is idempotent because the specification says servers should use the state values passed from the client to replace any existing values for the target resource. However, HTTP POST is non-idempotent since the HTTP spec states that POSTed values should be used to append to, not replace, an existing resource collection.)

In this case, a client application for our simple to-do service might need to access the list of available items, to filter that list, to view a single item, and to mark an item complete. Many of these actions use state values to pass data between the client and server. For example, the `add-item` action allows the client to pass the state values `title` and `dueDate`. Here is a diagram that illustrates those actions (previous page).

The actions shown in the diagram (and listed below) are also semantic descriptors -- they describe the semantic actions for this service.

- `read-list`
- `filter-list`
- `read-item`
- `create-item`
- `mark-complete`

As you work through the diagram, you might find that you missed actions or data items the client will want or need. That's an opportunity to go back to Step 1 to add new descriptors and/or improve on the diagram in Step 2.

Once you have reiterated through these two steps you should have a good idea of all the data points and actions the client will need to interact with your service.

Step 3: Reconcile magic strings

The next step is to reconcile all the "magic strings" in your service interface. The magic strings are all the descriptor names, which have no intrinsic meaning but merely represent actions or data elements that clients will access when communicating with your service. Reconciling these descriptor names means adopting well-known public names from sources like:

- Schema.org
- microformats.org
- Dublin Core
- IANA Link Relation Types

These are all repositories of well-defined, shared names. When you use these for your service interface, it is likely that developers will have seen these before and will understand what they mean. This can improve the usability of your API.

While it is a good idea to use shared names for descriptors on your service interface, you don't need to use them for your internal implementation (e.g. the data-field names in a database). The service itself can map the public interface names to the internal storage names without any problem.

For the sample to-do service, I was able to find acceptable existing names for all but one semantic descriptor: create-item. For this case, I resorted to creating a unique URI based on rules from the Web Linking RFC 5988. There are trade-offs to selecting well-known names for your interface descriptors. They rarely perfectly match your internal data-storage elements and that's OK.

Here are my results:

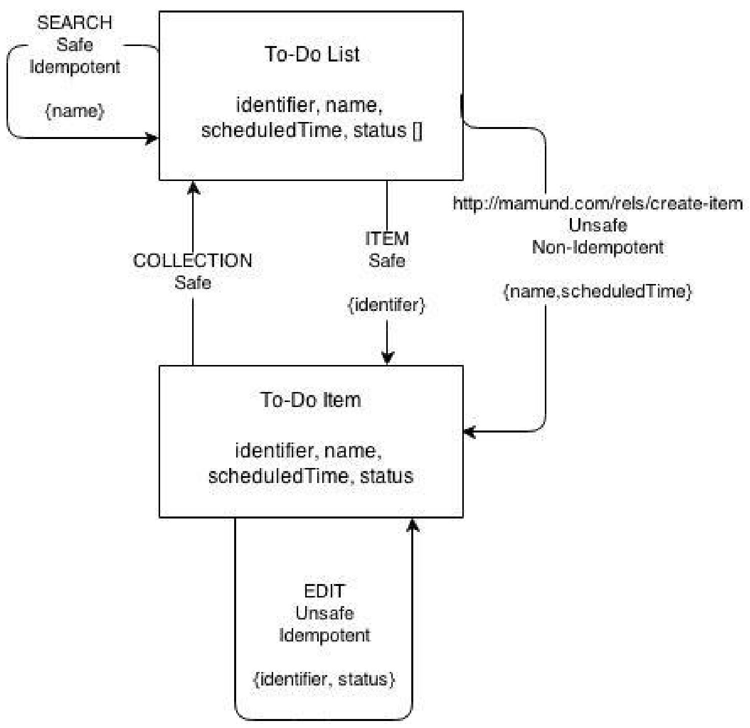
- `id -> identifier` from Dublin Core: http://purl.org/dc/elements/1.1/identifier
- `title -> name` from Schema.org: https://schema.org/name
- `dueDate -> scheduledTime` from Schema.org: https://schema.org/scheduledTime
- `complete -> status` from Schema.org: https://schema.org/status
- `read-list -> collection` from IANA Link Relation Types: http://www.iana.org/assignments/link-relations/link-relations.xhtml
- `filter-list -> search` from IANA Link Relation Types: http://www.iana.org/assignments/link-relations/link-relations.xhtml
- `read-item -> item` from IANA Link Relation Types: http://www.iana.org/assignments/link-relations/link-relations.xhtml
- `create-item -> http://mamund.com/rels/create-item` using RFC 5988
- `mark-complete - edit` from IANA Link Relation Types: http://www.iana.org/assignments/link-relations/link-relations.xhtml

Based on my name reconciliation, here is my updated state diagram: (see next page)

Step 4: Select a media type

The next step in the design process for your API is to select a media type to use when passing messages between client and server. One of the hallmarks of the Web is that data is passed as standardized documents over a uniform interface. It is important to select a media type that supports both the data descriptors (e.g. "`identifier`", "`status`", etc.) as well as the action descriptors (e.g. "`search`", "`edit`", etc.). There are quite a few formats available.

Some of the most popular hypermedia formats as I write this are (in no special order):



- Hypertext Markup Language (HTML)
- Hypertext Application Language (HAL)
- Collection+JSON (Cj)
- Siren
- JSON API
- Uniform Basis for Exchanging Representations (UBER)

It is also important to select a media type that will work well with your target protocol. Most developers prefer the HTTP protocol for service interfaces. However, WebSockets, XMPP, MQTT, and CoAP are also used, especially for high-speed, short-message, peer-to-peer implementations.

For this example, I'll use HTML as the message format and HTTP as the protocol. HTML has all the data-descriptor support that's needed (`<UL>` for lists, `<LI>` for items, and `<SPAN>` for data elements). It also has adequate support for action descriptors (`<A>` for safe links, `<FORM method="get">` for safe transitions, and `<FORM method="post">` for unsafe transitions).

(The state diagram currently shows the `edit` action as idempotent –ce.g. HTTP PUT but HTML still does not have native support for PUT. For this example, I'll use an added field to help make HTML's POST-only support idempotent.)

Now I can try out the interface by creating some sample representations based on the state diagram. For our example, we have only two representations to render: the "To-Do List" and the "To-Do Item" representations.

Remember, as you work through the representation samples of your state diagram, you may find things you missed in earlier steps (missing descriptors, changes in action descriptors such as idempotency, etc.). That's fine. Now is the time to work these all out –-before you commit this design to code.

Once you're satisfied that the representations are complete, there is an additional step you need to do before starting to write code: create the semantic profile.

Step 5: Create a semantic profile

A semantic profile is a document that lists all the descriptors in your design and includes details about each one to help developers when building both client and server implementations. The profile is an implementation guide, not an implementation description. This is an important distinction.

Service description formats

Service description document formats have been around for quite a while and are handy when you want to generate code for, or document, an existing implementation of a service. There are quite a few formats around.

The top contenders as I write the article are:

- Web Services Definition Language (WSDL)
- Atom Publishing Protocol (APP)
- Web Application Description Language (WADL)
- API Blueprint
- Swagger
- RESTful API Modeling Language (RAML)

Profile Formats

There are only a few profile formats at the moment. The ones I recommend are:

- Application-Level Profile Semantics (ALPS)
- JSON-LD and Hydra

Both are relatively new. The JSON-LD specification reached W3C Recommendation status early in 2014. Hydra is still an Unofficial Draft (as of this writing) and has an active community of developers. ALPS is still in early draft stage with the IETF.

Since a profile document should describe the real-life aspects of a problem space (not just a single implementation ▶

```html
<html>
  <head>
    <!-- for test display only -->
    <title>To Do List</title>
    <style>
      .name, .scheduledTime, .status, .item {display:block}
    </style>
  </head>
  <body>
    <!-- for test display only -->
    <h1>To-Do List</h1>

    <!-- to-do list collection -->
    <ul>
      <li>
        <a href="/list/1" rel="item" class="item">
          <span class="identifier">1</span>
        </a>
        <span class="name">First item in the list</span>
        <span class="scheduledTime">2014-12-01</span>
        <span class="status">pending</span>
      </li>
      <li>
        <a href="/list/2" rel="item" class="item">
          <span class="identifier">2</span>
        </a>
        <span class="name">Second item in the list</span>
        <span class="scheduledTime">2014-12-01</span>
        <span class="status">pending</span>
      </li>
      <li>
        <a href="/list/3" rel="item" class="item">
          <span class="identifier">3</span>
        </a>
        <span class="name">Third item in the list</span>
        <span class="scheduledTime">2014-12-01</span>
        <span class="status">complete</span>
      </li>
    </ul>

    <!-- search transition -->
    <form method="get" action="/list/" class="search">
      <legend>Search</legend>
      <input name="name" class="identifier" />
      <input type="submit" value="Name Search" />
    </form>

    <!-- create-item transition -->
    <form method="post" action="/list/" class="http://mamund.com/rel/create-item">
      <legend>Create Item</legend>
      <input name="name" class="name" />
      <input name="scheduledTime" class="scheduledTime" />
      <input type="submit" value="Create Item" />
    </form>

  </body>
</html>
```

“To-Do List” representation

```
<html>
  <head>
    <!-- for test display only -->
    <title>To Do List</title>
    <style>
      .name, .scheduledTime, .status, .item, .collection {display:block}
    </style>
  </head>
  <body>
    <!-- for test display only -->
    <h1>To-Do Item</h1>

    <a href="/list/" rel="collection" class="collection">Back to List</a>

    <!-- to-do list collection -->
    <ul>
      <li>
        <a href="/list/1" rel="item" class="item">
          <span class="identifier">1</span>
        </a>
        <span class="name">First item in the list</span>
        <span class="scheduledTime">2014-12-01</span>
        <span class="status">pending</span>
      </li>
    </ul>

    <!-- edit transition -->
    <form method="post" action="/list/1" class="edit">
      <legend>Update Status</legend>
      <input type="hidden" name="etag" value="q1w2e3r4t5y6" class="etag" />
      <input type="text" name="status" value="pending" class="status" />
      <input type="submit" value="Update" />
     </form>

  </body>
</html>
```

"To-Do Item" representation

within that space), the format is quite different than typical description formats.

You'll notice that this document looks like a basic vocabulary of all the possible data values and actions in the to-do service interface – and that's the idea. Services that agree to abide by this profile can make their own decisions about protocol, message format, and even URLs. Clients who agree to accept this profile will be built to recognize and, if appropriate, activate the descriptors shown in this document.

This is also a great format for generating human-readable documentation, analyzing similar profiles, tracking which profiles are most commonly used, and even generating state diagrams. But that's a subject for another article.

Now that you have the complete list of descriptors with reconciled names, the annotated state chart, and a semantic profile document, you're ready to start coding a sample server and client.

Step 6: Write some code

At this point, you should be able to turn over your design documents (state charts and semantic profile) to developers of both the server and client apps in order to start a specific implementation.

The HTTP server should implement the state diagram created in Step 2 and requests from the client should trigger the

```
<alps version="1.0">
  <doc>
    ALPS profile for InfoQ article on "API Design Methodology"
  </doc>

  <!-- data descriptors -->
  <descriptor id="identifier" type="semantic" ref="http://purl.org/dc/elements/1.1/identifier" />
  <descriptor id="name" type="semantic" ref="https://schema.org/name" />
  <descriptor id="scheduledTime" type="semantic" ref="https://schema.org/scheduledTime" />
  <descriptor id="status" type="semantic" ref="https://schema.org/status" />

  <!-- action descriptors -->
  <descriptor id="collection" type="safe" ref="http://www.iana.org/assignments/link-relations/link-relations.xhtml" />
  <descriptor id="item" type="safe" ref="http://www.iana.org/assignments/link-relations/link-relations.xhtml">
    <descriptor href="#identifier" />
  </descriptor>
  <descriptor id="search" type="safe" ref="http://www.iana.org/assignments/link-relations/link-relations.xhtml">
    <descriptor href="#name" />
  </descriptor>
  <descriptor id="create-item" type="unsafe" ref="http://mamund.com/rels/create-item">
    <descriptor href="#name" />
    <descriptor href="scheduledTime" />
  </descriptor>
  <descriptor id="edit" type="idempotent" ref="http://www.iana.org/assignments/link-relations/link-relations.xhtml">
    <descriptor href="#identifier" />
    <descriptor href="#status" />
  </descriptor>
</alps>
```

ALPS document for the to-do state chart

proper state transitions in the service. Each representation sent from the service should be in the format selected in Step 3 and should include a link to the profile created in Step 4. Responses should include the appropriate hypermedia controls that implement the actions shown in the state chart and described in the profile document. Client and server developers can build their implementations relatively independently at this point and use test runs to validate compliance with the state diagram and profile.

Once you have stable running code, there is one more step in our list: publishing.

Step 7: Publish your API

Web APIs should publish at least one URL that is promised to always respond to clients, even in the far future. I call this the "billboard URL" –-the one everyone knows. It is also a good idea to publish the profile document so that new implementations of the service can link to it in responses. You can also publish human-readable documentation, tutorials, etc. to help developers understand and use your service.

Once that is done, you should have a well-designed, stable, accessible service up and running, ready to use.

In conclusion

This article covered a set of steps for designing APIs for the Web, and focused on getting the data ►

and action descriptions correct and documenting them in a machine-readable way to make it easy for human developers to implement clients and servers for this design even if they are not in direct contact with each other.

The steps are:

1. **List all the Parts**
   Gather all the data elements clients will need in order to interact with the service

2. **Draw State Diagrams**
   Document all the actions (state transitions) that will be available for the service

3. **Reconcile Magic Strings**
   Clean up your public interface to match (as best possible) well-known names

4. **Select a Media Type**
   Review message formats to find the one that most closely aligns the service transitions with the target protocol.

5. **Create a Semantic Profile**
   Write up a profile document that defines all the descriptors used in the service

6. **Write Some Code**
   Share the profile document and the state diagram to client and server developers and start writing code to test compliance and adjust the profile/diagrams as needed.

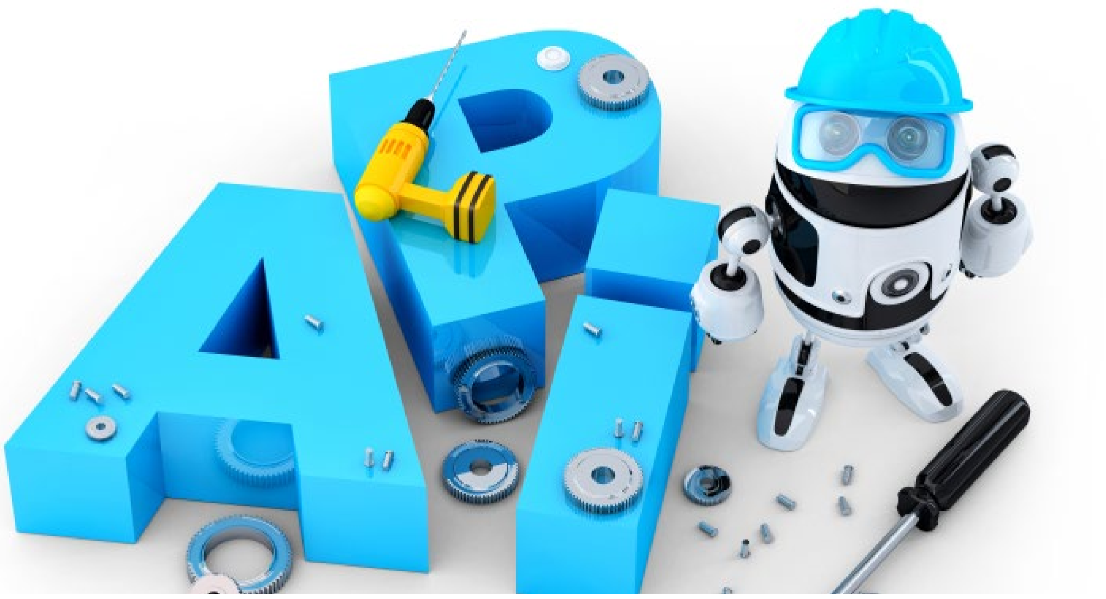
7. **Publish Your API**
   Publish your "billboard URL" and profile document so that others can use them to create new services and/or client applications.

It is likely that you'll need to reiterate through some steps along the way as you discover missing elements and make trade-off decisions with your design. The sooner that happens in the process, the better. It's also possible that you will be able to use this API design at some point in the future to create implementations using new formats and protocols that may be requested by developers at some point.

This methodology is just one possible way to create a dependable, repeatable, consistent process for designing your Web APIs. As you work through this example, you may find it works better for you to insert additional steps, collapse some, and, of course, the message format and protocol decisions may vary from one case to the next.

Hopefully, this gives you some ideas on how to create an optimal API design methodology for your organization or team. ■

Implementing Hypermedia



Steve Klabnik is a Rails committer, a Rust contributor, and the author of Rails 4 in Action, Designing Hypermedia APIs, and Rust for Rubyists. He currently lives in the New York area and travels the world speaking about the Web.

A lot of (virtual) ink has been spilled on the theory of using hypermedia in your APIs, but what about the practice? At the API Craft 2014 conference, many people who shared stories of using hypermedia sheepishly admitted that they hadn't talked about it publicly. Many of us resolved to share more of our stories. I'd like to share my own with you.

I'll talk about four real-world implementations of hypermedia: how you may already be using hypermedia through image links; how GitHub uses the Link header for pagination; using hypermedia in constrained systems like iOS; and how Balanced uses hypermedia principles to build its product. Each of these scenarios showcases a different aspect of using hypermedia in API design.

Thumbnail images

While much of the theory of hypermedia talks about hypermedia as the fundamental theory of your entire API, I have a little secret to share: it doesn't have to be that way. You can gain some of the advantages of hypermedia without entirely overhauling your API. The two cases I see most often concern thumbnail images and pagination.

You may be using hypermedia without even realizing it. I see this most often in APIs that contain images, often thumbnails. Consider this (partial) API response, from Twitter (Code 1).

Nobody would suggest that Twitter has a hypermedia API, but this response does indeed contain links and therefore is employing hypermedia. We could include these images inline using a data URI, but instead, we use links.

By including links, an API client is allowed to choose which information to download. The links inform the client of available options and where to

```
GET https://api.twitter.com/1.1/users/show.json?screen_name=rsarver

{
  "name": "Ryan Sarver",
  "profile_image_url": "http://a0.twimg.com/profile_images/1777569006/image1327396628_
normal.png",
  "created_at": "Mon Feb 26 18:05:55 +0000 2007",
  "location": "San Francisco, CA",
  "profile_image_url_https": "https://si0.twimg.com/profile_images/1777569006/
image1327396628_normal.png",
  "utc_offset": -28800,
  "id": 795649,
  "lang": "en",
  "followers_count": 276334,
  "protected": false,
  "profile_background_image_url_https": "https://si0.twimg.com/profile_background_
images/113854313/xa60e82408188860c483d73444d53e21.png",
  "verified": false,
  "time_zone": "Pacific Time (US & Canada)",
  "description": "Director, Platform at Twitter. Detroit and Boston export. Foodie
and over-the-hill hockey player. @devon's lesser half",
  "profile_background_image_url": "http://a0.twimg.com/profile_background_
images/113854313/xa60e82408188860c483d73444d53e21.png",
```

Code 1

find them. In other words, this is just as "real" a form of hypermedia as any other usage.

To make these benefits a bit more clear, let's consider a slightly different response:

```
GET https://api.example.com/profile

{
  "name": "Steve",
  "picture": {
    "large": "https://somecdn.com/
pictures/1200x1200.png",
    "medium": "https://somecdn.com/
pictures/100x100.png",
    "small": "https://somecdn.com/
pictures/10x10.png"
  }
}
```

We introduce hypermedia here to avoid including all three versions of the profile image. We tell our clients that there are three possible images available, and we tell the client where it can find each image. Our client is now able to choose what it wants to do based on what it's trying to accomplish in the moment. It does not have to download all three versions of the image if it only wants one. We've made our payload smaller, we've increased client flexibility, and we've increased discoverability.

What I'm getting at here is that you may already be deploying a teeny bit of hypermedia even if you've never thought about it before. And you didn't need to design your whole API around hypermedia to gain the benefit in this one case.

Pagination

Pagination is another area where a tiny bit of hypermedia can considerably simplify client code. Let's take GitHub as a real-world example of this. In its documentation, GitHub talks about one of the constraints of its API:

Different API calls respond with different defaults. For example, a call to list GitHub's public repositories provides paginated items in sets of 30, whereas a call to the GitHub Search API provides items in sets of 100.

It's easier to communicate the default when the response is inline. Let's examine how GitHub actually implements this.

You make a request to a paginated resource, such as their search resource:

```
GET "https://api.github.com/search/
  code?q=addClass+user:mozilla"
```

It returns a Link header:

```
Link: <https://api.github.com/search/
  code?q=addClass+user%3Amozilla&
  page=2>; rel="next",
   <https://api.github.com/search/
  code?q=addClass+user%3Amozilla&
  page=34>; rel="last"
```

The Link header, defined in RFC 5988, gives us, well, links. The links consist of a URL and a link ►

relation, which is where the rel comes from. Because we're on the first page of the results, GitHub shows us that we have a next and last option.

If we fetch the link at next, we get a different set of headers:

```
Link: <https://api.github.com/search/
    code?q=addClass+user%3Amozilla&page=15>;
    rel="next",
<https://api.github.com/search/
    code?q=addClass+user%3Amozilla&page=34>;
    rel="last",
<https://api.github.com/search/
    code?q=addClass+user%3Amozilla&page=1>;
    rel="first",
<https://api.github.com/search/
    code?q=addClass+user%3Amozilla&page=13>;
    rel="prev"
```

Now, we can see that there's also a prev and first page, too.

So where's the advantage? Well, client code is easier. Consider Ruby. If we wanted to get the next page of search results in the traditional manner, we'd do this:

```
require 'uri'
url = "https://api.github.com/search/
    code"
per_page = 15
current_page = 1
next_page = 1 # zero based, of course
page = (current_page + next_page *
    per_page).to_s

query = "addClass+user%3Amozilla"

uri = URI(url)
uri.query = URI.encode_www_
    form([["q", query], ["page", page]])

puts Net::HTTP.get(uri);
```

With hypermedia, we instead do this:

```
# response contains parsed body from
    previous request.

puts Net::HTTP.get(response.
    headers[:link].rels[:next])
```

It's much easier and far less prone to errors. Furthermore, if GitHub decides to change the defaults to 10 per page and forbids 15 per page, the hypermedia code will not need to change. The first one will, and until you've fixed the bug, your users will be stranded.

iOS hypermedia

One growth area I see for hypermedia is iOS. Here's why: in order to make changes to an iOS app, you have to go through Apple's approval process. But if you use hypermedia, the server can change the behavior of the client. Long ago, some friends and I did this for a project. Names have been changed to protect the innocent....

We were working on a podcast application. As such, we served large audio and video files. At the time, Apple had a restriction on audio: you could not serve high-quality audio over the GSM connection. We devised a scheme to get around this: links.

When the app was under review, we would have our server serve the podcast with low-quality audio links. The review would pass, and then we would have the server serve high-quality audio. Our customers would then get a free upgrade to higher-than-technically-allowed files. Sneaky!

Once we had this idea, however, we applied it to other aspects of the application. For example, some podcasts would also broadcast live and allow you to call into the show. In the UI, we made the app ask the server if the show was broadcasting live or not. If it was, the app displayed a button you could click to call in. The app would poll this endpoint as long as you were on the screen, and once the show ended, it would disable the button. This kind of server-driven interaction is hypermedia's bread and butter, but would be impossible if we had to redeploy a new client to change the state of the button.

The people who ran the podcast could provide the app with information about when the podcast would air. Something like "Each Friday at noon" would appear on an information screen. The app fetched this information from our server. If the podcasters wanted to change the time of the show on an app that didn't fetch the information from the server, app users would need an app update in order to receive the corrected information and the updated app would have to go through the "waiting for review" process again. Because the profile was server-driven, as soon as the show operator would click "save", every app would automatically retrieve that new information.

This involves hypermedia in two ways. First, the mentality that the server dictates the possibilities and the client displays those possibilities is central to the hypermedia way of API design. Each of these three instances is a great example of this principle in action.

Second, we implemented this through links. Upon startup, the app would fetch a configuration XML file from the server, which would provide a link to the RSS feed, a link to the "find out if we're on the air" feed, and a link to the profile information. The app would then use these links as appropriate. To implement the RSS switch, we'd just change feed at the link destination from low quality to high quality, and the app would fetch that entirely different feed by default.

There's a lot of fruitful ground to be explored here. A client that reacts to what a server says without client updates is crucial when you cannot often update the client. In situations like iOS or embedded devices, this constraint is obvious, and your users probably do not update your client as often as you'd like....

Case study: Balanced

The API of Balanced, my previous employer, is hypermedia enabled and follows the JSON API standard that I co-author. My previous examples described the addition of a sprinkling of hypermedia to responses, but Balanced is fully hypermedia-driven. This has led to good effects but also to some challenges.

On the good side, new features can roll out without breaking older clients. For example, the 1.1 release of the API was the first to completely follow the JSON API spec. After 1.1 was released, Balanced launched a Push to Card feature, which was entirely new. Because of hypermedia, the company did not need to release this feature as API version 1.2: older clients simply ignored the new feature, and new clients were able to use it. This makes operations significantly easier, as having many different versions complicates both deployment and development. This trend will continue as Balanced continues to add features to its API.

Hypermedia enthusiasts often talk about the benefits, but it's not all positive. In the interest of balance, (pun intended), I'd like to mention one of the downsides. When a customer reports a support issue, the first question you need to ask them is "Who are you?" In many cases, that would be "What's your customer ID?" Since Balanced uses hypermedia, it doesn't have a customer ID: it has a customer URL. Occasionally, customers would be confused when we'd ask for their customer URL.

This kind of thing will change as more people understand hypermedia APIs, but because we're in the early days, anyone who creates an API that's fully hypermedia-driven needs to be willing to help educate users on how to use it. In Balanced's case, this meant providing clients for many different languages up front, because many people don't know how to develop good hypermedia clients yet. While it's a good idea to give your customers pre-built clients anyway, in this case, Balanced had to, whereas with a more conventional API, they could have made a business decision to focus development efforts elsewhere.

Conclusion

As you can see, hypermedia can take many different forms, and doesn't have to be the sole organizing principle of your API. First, we talked about how you may be using hypermedia without realizing it, via image links. Then, we talked about GitHub and its pagination example. Next, we went over how a server-driven client doesn't need to be updated as often, which helps in constrained environments like iOS. Finally, we talked about a company that has used hypermedia as a competitive advantage, but not without a drawback or two.

I hope that these real-world implementations of hypermedia help you realize that hypermedia doesn't have to be all or nothing, and that you may already be doing it in some cases. I'm excited to see hypermedia spring up in more and more APIs, and to hear people talk about their successes and failures with the technique. ■

You may be using hypermedia without even realizing it.

- Steve Klabnik

Roy Fielding on Versioning, Hypermedia, and REST

by Mike Amundsen

Roy Fielding is currently senior principal scientist at Adobe Systems. While a graduate student at University of California, Irvine, he worked on a class project to create a maintenance robot for the Web called MOMSpider and created the libwww-perl library based on Tim Berners-Lee's libwww. There, he derived some of the underlying principles behind the architecture of the WWW, calling it the HTTP Object Model and later renaming it Representational State Transfer or REST.

Roy Fielding is a major force in the world of networked software. Roy's contributions to open standards are extensive. His name appears on more than a dozen RFC specifications including HTTP, URI Templates, and others. Roy is also one of the editors for the W3C's Do Not Track standards effort. As a founding member of the Apache Software Foundation, he was instrumental in the creation of the Apache HTTP Server (httpd) server along with a number of other open-source projects.

Roy took some time while traveling between standards meetings to answer InfoQ's questions on a topic that often starts debates: versioning on the Web. He also talked about why hypermedia is a requirement in his REST style, the process of designing network software that can adapt over time, and the challenge of thinking at the scale of decades.

In August 2013, you gave a talk at the Adobe Evolve conference in which you offered advice on how to approach versioning APIs on the Web. It was a single word: "Don't." What reaction to that guidance have you seen?

I think everyone in attendance had a positive reaction, since most are our customers and familiar with the design rationale behind the Adobe Experience Manager products. Of course, I wasn't *reading* the slides for that audience. I was explaining the *rationale* behind the conclusions seen in them.The Internet reaction to the published slides was a little more mixed, with some folks misunderstanding what I meant by versioning and others misunderstanding the point about changing the hostname/branding. By versioning, I meant sticking client-visible interface numbers inside various names so that the client labels every interaction as belonging to a given version of that API.

Unfortunately, versioning of interface names only manages change for the API owner's sake. That is a myopic view of interface design, one where the owner's desire for control ignores the customer's need for continuity.What happens when you version an API?

Either, (a) the version is eventually changed and all of the components written to the prior version need to be restarted, redeployed, or abandoned because they cannot adapt to the benefits of that newer system, or (b) the version is never changed and is just a permanent lead weight making every API call less efficient.

A lot of developers throw up their hands in disgust at this point and claim that I just don't understand their problem. Their systems are important. They are going to change. New features are going to be provided. Data is going to be rearranged. They need some way to control how old clients can coexist with new ones.

Naturally, that is where I have to explain why "hypermedia as the engine of application state" is a REST constraint. Not an option. Not an ideal. Hypermedia is a constraint, as in you either do it or you aren't doing REST. You can't have evolvability if clients have their controls baked into their design at deployment. Controls have to be learned on the fly. That's what hypermedia enables.

But that alone is still not enough for evolvability. Hypermedia allows application controls to be supplied on demand, but we also need to be able to adapt the clients' understanding of representations (understanding of media types and their expected processing). That is where code-on-demand shines.

So, one of the reasons hypermedia is a requirement in the REST style is to deal with change over time?

Anticipating change is one of the central themes of REST. It makes sense that experienced developers are going to think about all of the ways that their API might change in the future, and to think that versioning the interface is paving the way for those changes. That led to a never-ending debate about where and how to version the API.

The techniques that developers learn from managing in-house software, where they might reasonably believe they have control over deployment of both clients and servers, simply don't apply to network-based software intended to cross organizational boundaries. This is precisely the problem that REST is trying to solve: how to evolve a system gracefully without the need to break or replace already deployed components.

Hence, my slides try to restore focus to where it belongs: evolvability. In other words, don't build an API to be RESTful, build it to have the properties you want. REST is useful because it induces certain properties that are known to benefit multi-org systems, like evolvability. Evolvability means that the system doesn't have to be restarted or redeployed in order to adapt to change.

Does that mean that as long as I use the REST style, I am free and clear of versioning issues?

No. It is always possible for some unexpected reason to come along that requires a completely different API, especially when the semantics of the interface change or security issues require the abandonment of previously deployed software. My point was that there is no need to anticipate such world-breaking changes with a version ID. We have the hostname for that. What you are creating is not a new version of the API but a new system with a new brand.

On the Web, we call that a new Web site. Web sites don't come with version numbers attached because they never need to. Neither should a RESTful API. A RESTful API (done right) is just a Web site for clients with a limited vocabulary.

One of the things you say about REST is that it was designed to support "software engineering on the scale of decades." What does that mean, precisely?

REST was originally created to solve my problem: how do I improve HTTP without breaking the Web? It was an important problem to solve when I started rewriting the HTTP standard in 1994-95. I was a post-Masters Ph.D. student in software engineering, trying not to screw up what was clearly becoming the printing press of our age, which means I had to define a system that could withstand decades of change produced by people spread all over the world. How many software systems built in 1994 still work today? I meant it literally: decades of use while the system continued to evolve, in independent and orthogonal directions, without ever needing to be shut down or redeployed. It's two decades, so far.

A RESTFUL API (DONE RIGHT) IS JUST A WEB SITE FOR CLIENTS WITH A LIMITED VOCABULARY.

- Roy T. Fielding

You have acknowledged that this is a level of engineering at which most architects, designers, and developers don't operate. So why talk about this level of engineering scale?

I talk about it because the initial reaction to using REST for machine-to-machine interaction is almost always of the form "We don't see a reason to bother with hypermedia – it just slows down the interactions, as opposed to the client knowing directly what to send." The rationale behind decoupling for evolvability is simply not apparent to developers who think they are working towards a modest goal, like "works next week" or "We'll fix it in the next release."If developers can conceive of their systems being used for a much longer time, then they can escape their own preconceptions about how it will need to change over time. We can then work back from decades to years (How long until you don't know your users?) or even months (How long until you've lost control over client deployment?).

The HTTP application-level protocol is often cited as an example of successful engineering at the scale of decades. Yet, HTTP has gone through more than one version and the early versions of HTTP got a number of things wrong including the host header problem, absolute time-caching directives, and others. Does that run counter to your "Don't" guidance for Web APIs?

No, HTTP doesn't version the interface names – there are no numbers on the methods or URIs. That doesn't mean other aspects of the communication aren't versioned. We do want change, since otherwise we would not be able to improve over time, and part of change is being able to declare in what language the data is spoken. We just don't want breaking change. Hence, versioning is used in ways that are informative rather than contractual.

By the way, it is more accurate to say that HTTP got almost everything right, but that the world simply changed around (and because of) it. Host would have been a stupid idea in 1992 because nobody needed multiple domains per IP until the Web made being on the Internet a business imperative. Persistent connections would have been a terrible idea up until Mosaic added embedded images to HTML. And absolute times for expiration made more sense when people hosting mirrors looked at those fields, not caches, and the norm was to expire in weeks rather than seconds.

What lessons can we draw from the fact that HTTP and even HTML have changed over time?

What we learned from HTTP and HTML was the need to define how the protocol/language can be *expected* to change over time, and what recipients ought to do when they receive a change they do not yet understand. HTTP was able to improve over time because we required new syntax to be ignorable and semantics to be changed only when accompanied by a version that indicates such understanding.

It seems Web developers struggle to handle change more than in the past. Are we running into new problems? Or just seeing more of the same?

I think there are just more opportunities to struggle now than there were in the past. It has become so easy for people to create systems with astonishing reach, whereas it used to take years to get a company just to deploy a server outside its own network. It's a good problem to have, most of the time.

Software developers have always struggled with temporal thinking.

Finally, in addition to "Don't," what advice you would pass on to Web API designers, architects, and developers to help them deal with the problem of change over time?

I didn't say don't change over time – just don't use deliberately breaking names in an API.

I find it impossible to give out generic device, since almost anything I could say would have to be specific to the context and type of system being built. REST is still my advice on how to build an application for the Web in a fashion that is known to work well over time and known to create more Web as a result (more addressable resources). ■

REST-y Reader

Mike Amundsen in his role of Director of Architecture for the API Academy, Amundsen heads up the API Architecture and Design Practice in North America. He 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" published in 2013. Mike's 2011 book, "Building Hypermedia APIs with HTML5 and Node", is an oft-cited reference on building adaptable Web applications.

I am regularly asked which books I recommend for those who want to learn more about designing, implementing, and maintaining APIs for the Web. Here's a short list that covers quite a bit of ground with a minimum amount of reading. These are books I found along the way as I was learning about APIs and many of them are still on my go-to shelf that holds the book I consult most often.

The primary list is a handful of books that speak directly to the work of HTTP, APIs, REST, and hypermedia. These are not the only books on the subjects but they are the ones I find myself referring to most often in my own work.

The secondary list contains books that, while not directly in the field of APIs, have affected my thinking on the way we design and implement stuff on the Web. I had a hard time narrowing down this list and there are quite a few more I'd add, but I'll save that for another time.

Finally, I added a section for other resources. These are useful sources that are not book length. Mostly these are blog posts, peer-reviewed papers, etc. that cover topics that come up when working through a tough problem or trying to get a handle on the concepts behind common practice and theory. I often keep bookmarks to these items close at hand.

Primary books

RESTful Web Services

Leonard Richardson and Sam Ruby (2008)

This essential book was one of the first to document the create-read-update-delete (CRUD) style of HTTP APIs – a style that is still the most common way to create RESTful APIs. The writing style is excellent and the material top-notch. It's also cool that publisher O'Reilly Media has released this under the Creative Commons License in several e-book formats. You should definitely have this one handy.

HTTP Developer's Handbook

Chris Shiflett (2003)

I found this book an excellent guide and reference for the HTTP protocol and for writing programs that use HTTP. I still refer to this book when trying to grok HTTP edge cases. I still love this well-written and organized book.

RESTful Web Services Cookbook

Subbu Allamaraju (2010)

This is a great set of simple recipes for solving real-world problems for HTTP APIs. While this book covers the usual things like designing URIs, payloads, and handling status codes, etc., I really like its recipes for supporting async operations, long query strings, and what Allamaraju calls "HTTP controllers". I'm on my second copy and it's already well worn.

REST in Practice

Jim Webber, Savas Parastatidis, and Ian Robinson (2010)

This book has a distinct enterprise feel and was one of the first to illustrate the use of hypermedia for business applications. Its sample app for RESTBucks coffee shop is an oft-cited example. With sample code in both C# and Java, this book speaks to common enterprise developers as well as to those interested in what it is like to mount a full-featured app from start to finish.

Building Hypermedia APIs with HTML5 and Node

Mike Amundsen (2011)

I wrote this book to explore the parts of the REST architecture that Fielding had left out: the details behind "hypermedia as the engine of application state" (HATEOAS). It's a very short book that covers sample hypermedia designs in XML, JSON, and HTML. I've received lots of positive comments on the book and often find its references and examples showing up online and in customers' internal materials.

REST API Design Rulebook

Mark Masse (2011)

This book is a good source of design patterns for CRUD-style APIs. It covers the basics like URI design for singulars, plurals, and operations, and decently reviews the HTTP spec for methods, headers, and status codes. For me there is a bit too much reliance on Masse's own WRML format but that is just my personal preference.

RESTful Web APIs

Leonard Richardson and Mike Amundsen (2013)

This book is the sequel to Richardson and Ruby's *RESTful Web Services* and covers a lot of new territory. Where the earlier book focused on HTTP resources, this one emphasizes hypermedia formats. It dips a toe into the Semantic Web waters with reviews of a handful of RDF-based formats and introduces profiles to help carry application-level meaning into the Web API space. It was fun for me to work with Leonard Richardson on this and we have gotten lots of great feedback.

Secondary books

I read quite a few books and find some that are technically outside the field of APIs, HTTP, and REST but that are still quite helpful when I am working on designing and implementing distributed apps. Here are a few of my top picks for expanding your mind outside the typical API readings.

The Design of Everyday Things

Don Norman (2013)

Originally published in 1988, this book sets the groundwork for the field of human-computer interaction (HCI) and what we know as usability. Norman works through all sorts of examples of how humans and devices (not just computers) interact. The book defines his action lifecycle, clarifies the notion of affordance, and gives some excellent advice on how to think about designing user interfaces based on your audience, the environment in which the device is used, and the goal you have in mind as a designer. This is a must-have for your bookshelf.

In Search of Certainty

Mark Burgess (2013)

Burgess is the creator of the CFEngine technology, which acts as an independent agent that monitors large-scale infrastructure. His experience led him to apply an immunology model to complex computer systems, and the book chronicles his journey from a basic idea to a full set of tools with their own agency on the network. I like the writing style and the great references to physics, brain science, and the history of computing in general. Toward the end, Burgess puts in a bit more reference to his product than I'd prefer, but it makes sense since it is the work on CFEngine that brings all these themes together. This book gets you thinking about what it takes to create safe and successful autonomous bots on the network.

Information: A History, a Theory, a Flood
James Gleick (2012)
Gleick's 1987 book *Chaos* popularized the concepts in chaos theory and this book does the same for information theory. Gleick has a great storytelling style and covers quite a bit of ground in the book's 300+ pages. He covers it all, from remote drumming to quantum computing, with excellent stories and real insight. I still go back and read sections of this book just for the enjoyment.

Information: A Very Short Introduction
Luciano Floridi (2010)
This handy text supplies the basics of information theory. It starts with a clear explanation on how data is different than information and talks about how information theory is tied up on math, physics, biology, economics, and even social ethics. Where Gleick tells stories, Floridi lays out the facts and links between fields of study. These two books make a great pair.

Memory Machines: The Evolution of Hypertext
Belinda Barnet (2013)
If you want to learn how hypertext morphed and grew over the last 50 years (yes, one half of a century) then this is book for you. Barnet has collected personal interviews with Ted Nelson, Douglas Engelbart, Tim Berners-Lee, and many others. You get a picture of how competing ideas and market forces shaped the modern hypertext/hypermedia world. You also learn about many initial ideas on the use and implementation of hypertext that are still not yet available.

Complexity: A Guided Tour
Melanie Mitchell (2011)
This accessible book on the world of complexity theory takes the reader through a bit of history, genetic programming theory for devices, and automata programming for large systems. Along the way, there are clear examples and even references to running code. Mitchell's work is an excellent text for anyone interested in digging into this huge topic.

Other sources

Here are papers, blog posts, and other sources I keep handy for review and reference. These are the links I often send to others when they ask for examples, clarification, or background on a topic.

Architectural Styles and the Design of Network-Based Software Architectures
Roy T. Fielding (2000)
This is the dissertation that launched a thousand arguments. Seriously, this is a document everyone in the API space should read. It's not long and it's relatively easy to read. And it is the document that defines REST. Often, if people read this at all, they only read the fifth chapter, "Representational State Transfer (REST)". To me, that is not the best part of the dissertation. Instead, I like the second chapter, "Network-Based Application Architectures", in which Fielding describes the rationale and desired outcome for building distributed systems. In fact, I recommend reading this dissertation in the following order: chapters 2, 1, 4, 3, 6, and 5. That's right, read the REST chapter last. I think it makes a lot more sense when you do this.

"REST APIs must be hypertext-driven"
Roy T. Fielding: *Untangled* (2008)
The same year *RESTful Web Services* was released, Fielding published this now (in)famous blog post explaining that hypermedia (he uses the word hypertext) is a required element for his REST style. In another important step, he uses this post to expand on what he meant by the phrase "hypertext as the engine of application state" – what people have gotten into the habit of calling HATEOAS and which Fielding prefers to call the "hypermedia constraint". The comment thread below this post is one of the most valuable conversations I've seen Fielding engage in over the topic of REST.

"*What's different/better/worse than other JSON hypermedia media types?*"
Kevin Swiber et al: *GitHub* (2013)
In mid-2013, Emmanuel Gomez asked in Kevin Swiber's Siren JSON hypermedia format repo a great question about how to choose which hypermedia JSON format to use for implementation. The subsequent comment thread is pure gold. Most of the authors of these popular formats (Mike Kelly for HAL, Kevin Swiber for Siren, Jorn Wildt for MASON, myself for Collection+JSON) chime in, and there are a number of insights in the thread. If you're contemplating a JSON format for hypermedia, this is a great resource of commentary.

Hypermedia Types
Mike Amundsen (2011)
This is a series of Web pages I started in 2010 in order to analyse and categorize hypermedia formats based on a set of properties that all hypermedia types will exhibit. A set of elements called "H-Factors" ▶

are defined to help study how hypermedia actually works for inline business applications and this has been picked up by several other people to help in both design and analysis of hypermedia formats. It's a work in progress and I sometimes pass this link to folks asking about some fundamentals of what hypermedia controls are and how they work.

"How to Follow Instructions"
Leonard Richardson: *InfoQ* (2012)
In this presentation, Richardson discusses how hypermedia can help create machine-readable instructions for clients to follow. He adopts a helpful point of view (and recalls the days of BBSs and file servers before HTTP and links were common) and the slides are excellent, too. It's well worth the 50 minutes.

Designing Hypermedia APIs
Steve Klabnik (2014)
Klabnik is a great resource for down-to-earth, no-nonsense advice on what it takes to actually implement running code that uses hypermedia. He started this living book in 2012 and continues to add to it. If you're looking for a complete story, cover to cover, this is probably not a good resource for you. However, if you want to keep an eye on the ever-growing body of knowledge on what it is like to actually build and maintain these types of systems, *Designing Hypermedia APIs* is a great addition to your regular reading. I always look forward to his periodic updates.

Conclusion

That's a tour through my top-level bookshelf and my common bookmarks. There are quite a number of other books and links worth sharing but this is a good start. If you have other sources you use often when you are designing and implementation Web APIs, I'd love to hear about them. ■

CONTINUOUS DELIVERY STORIES
eMag Issue 21 - December 2014
InfoQ
THIS ISSUE Adopting Continuous Delivery
ARTICLE Delivering Continuous Delivery Continuously
INTERVIEW Dave Farley on Continuous Delivery
INTERVIEW Rachel Laycock on Continuous Delivery

INFRASTRUCTURE CONFIGURATION MANAGEMENT TOOLS
InfoQ

Automation in the Cloud and Management at Scale
InfoQ
Lessons on Cloud Management at Scale: An Interview With Randy Skopecek

Agile Project Management
InfoQ
Why the Agile Project Manager is the Secret Sauce for Development Projects