Web APIs with Python Coursepack

Stephen Zweibel, Mary Catherine Kinniburgh, Jonathan Reeve, and Patrick Smyth

May 14, 2019

Contents

Part I

Course Plan

Description

This course is aimed at humanities scholars interested in tapping into the data streams and functionality offered by platforms and content providers such as Twitter, Google, and the New York Times. Introduction to APIs will open with the basics of Python, a scripting language widely used in industry and the academy because of its human readability. We will proceed to the fundamentals of working with Application Programming Interfaces (APIs), the most common way to programmatically access web-based services and data. Lessons will cover the fundamentals of programming, the workflow of building a small script/app, accessing data from a variety of sources, and reading technical documentation. The course will be useful for those interested in understanding programming concepts, developing applications, and working with data.

This is a hands-on course. Consider this offering in complement with Fundamentals of Programming, CloudPowering DH Research, Practical Software Development for Nontraditional Digital Humanities Developers, or Introduction to Data for Digital Humanities Projects; and more!

Objectives

Students will come away from this course having learned how to:

- Work with web APIs and understand their structure.
- Use the command line.
- Work in Python at a fundamental level.
- Read and interpret technical documentation.
- Explore their own research questions using these techniques.

Before DHSI

To prepare for this course, please:

1. Review the below readings.
2. Consider the list of APIs, and think of which you might like to use.
3. Bring an idea or two for projects or research you would like to pursue.

What can you do with data from an API?

- Write a program that finds a book that was published on this day 100 or 200 years ago
- Find the most recent movie where all the actors are dead
- Write a program that finds the weather in your neighborhood and suggests movie based on keywords
- Write a program that gets a recent news topic and automatically finds a gif for it from giphy.com.
- Write a program that translates a news headline into Yoda-speak.

API Examples

- Chronicling America - Historical American newspaper data
- Corpus DB - Full texts and metadata from public domain books. Under development by Jonathan Reeve.
- Marvel API - Metadata on Marvel comics, movies, and TV shows.
- Twitter API - Extract tweet data and control Twitter accounts automatically.
- New York Times APIs - Work with New York Times articles, access book lists, or use tagging metadata to get info on people, places, and concepts.
- Debates in the Digital Humanities - Extract text and comments from Debates.
- APIs for Scholarly Resources - A list of useful APIs for researchers.

Schedule

Day One	10:15-12:00	Ice Breaker
		Introducing APIs
	1:30-2:30	Intro to Web
		What is HTTP?
	2:30-2:45	Break
	2:45-4:00	Researching APIs
Day Two	9:00-12:00	Class presentations
	10:45-12:00	Command Line and cURL
	12:00-1:30	Lunch
	1:30 - 2:45	Python I
	2:45-3:00	Break
	3:00-4:00	Research Question Workshop
Day Three	9:00-12:00	Python II
		Accessing APIs
		Data Formats
	12:00-1:30	Lunch
	1:30-4:00	Reading API Documentation
		Anatomy of a URL
Day Four	9:00-12:00	Manipulating Data
		Cleaning Data
		Exploring Data
	12:00-1:30	Lunch
	1:30-4:00	Project Lab
Day Five	9:00-10:30	Workshop
	10:45-12:00	Group Presentation
	12:00-1:30	Lunch
	1:30-4:00	DHSI Presentation

Part II

Course Readings

Reading 1

Do APIs Have a Place in the Digital Humanities?

Dan Cohen

is a Vice Provost, Dean, and Professor at Northeastern University

Do APIs Have a Place in the Digital Humanities?

APIs, Google, Programming, Yahoo November 21, 2005

Since the 1960s, computer scientists have used application programming interfaces (APIs) to provide colleagues with robust, direct access to their databases and digital tools. Access via APIs is generally far more powerful than simple web-based access. APIs often include complex methods drawn from programming languages—precise ways of choosing materials to extract, methods to generate statistics, ways of searching, culling, and pulling together disparate data—that enable outside users to develop their own tools or information resources based on the work of others. In short, APIs hold great promise as a method for combining and manipulating various digital resources and tools in a free-form and potent way.

Unfortunately, even after four decades APIs remain much more common in the sciences and the commercial realm—for example, the APIs provided by search behemoths Google and Yahoo—than in the humanities. There are some obvious reasons for this disparity. By supplying an API, the owners of a resource or tool generally bear most of the cost (on their taxed servers, in technical support and staff time) while receiving little or no (immediate) benefit. Moreover, by essentially making an end-run around the common or "official" ways of accessing a tool or project (such as a web search form for a digital archive), an API may devalue the hard work and thoughtfulness put into the more public front end for a digital project. It is perhaps unsurprising that given these costs even Google and Yahoo, which have the financial strength and personnel to provide APIs for their search engines, continue to keep these programs hobbled—after all, programmers can use their APIs to create derivative search engines that compete directly with Google's or Yahoo's results pages, with none of the diverting (and profitable) text advertising.

So why should projects in the digital humanities provide APIs, especially given their often limited (or nonexistent) funding compared to a Google or Yahoo? The reason IBM conceived APIs in the first place, and still today the reason many computer scientists find APIs highly beneficial, is that unlike other forms of access they encourage the kind of energetic and creative grass-roots and third-party development that in the long run—after the initial costs borne by the API's owner—maximize the value and utility of a digital resource or tool. Motivated by many different goals and employing many different methodologies, users of APIs often take digital resources or tools in directions completely unforeseen by their owners. APIs have provided fertile ground for thousands of developers to experiment with the tremendous indices and document caches maintained by Google and Yahoo. New resources based on these APIs appear weekly, some of them hinting at new methods for digital research, data visualization techniques, and novel ways to data-mine texts and synthesize knowledge.

Is it possible—and worthwhile—for digital humanities projects to provide such APIs for their resources and tools? Which resources or tools would be best suited for an API, and how will the creators of these projects sustain such an additional burden? And are there other forms of access or interoperability that have equal or greater benefits with fewer associated costs?

Reading 2

A Critical Reflection on Big Data

First Monday, Volume 18, Number 10 - 7 October 2013

PEER-REVIEWED JOURNAL ON THE INTERNET

A critical reflection on Big Data: Considering APIs, researchers and tools as data makers

by Farida Vis

Abstract

This paper looks at how data is 'made', by whom and how. Rather than assuming data already exists 'out there', waiting to simply be recovered and turned into findings, the paper examines how data is co–produced through dynamic research intersections. A particular focus is the intersections between the application programming interface (API), the researcher collecting the data as well as the tools used to process it. In light of this, this paper offers three new ways to define and think about Big Data and proposes a series of practical suggestions for making data.

Contents

1. Introduction

In the summer of 2012, Twitter CEO Dick Costolo, during an interview on NBC's *Today* morning show, declared that he had figured out what the platform was all about, stating that: 'Twitter brings you closer' (Costolo in Wasserman, 2012). Later, at the Wired Business Conference he elaborated further, highlighting that there was a common shape to so many of the different stories on Twitter. Costolo emphasised how stories can be observed both from a distance, giving one perspective, as well as from a much closer, more personal vantage point, bringing you closer. To explain his point more elaborately, he went on to give a detailed description of the 2010 art installation in Tate Modern's Turbine Hall by the Chinese artist (and fêted Twitter celebrity) Ai Weiwei. Costolo describes the installation as follows:

> '[Weiwei] created this installation that was at the Tate museum in London a while back and the installation was these hundreds of thousands of ceramic hand painted sunflower seeds. And as you stood back from the room it looked like this sea of just stones that were black stones that were spread across the floor and of course you couldn't really tell what they were. But as you got closer it looks like ... you can start to tell 'ooh it looks like they've stamped out hundreds of thousands of sunflower seeds and spread them across the floor'. But as you pick them up you started to realise that they were all individually shaped and painted differently and unique and beautiful and distinct in their own right. *So that's what we want to bring to what we're building: the ability to shrink the world and allow everybody to see each other*.' (Emphasis added, Costolo in Vis, 2012a).

It can be assumed that by drawing on this catchy and accessible visual metaphor Costolo sought to render the Twittersphere both visible and comprehensible. What we can see, or rather imagine, by way of the art installation is the enormous volume of tweets that are produced daily on the platform. One billion tweets are now sent every two and a half days, making it hard data to deal with in retrospect, never mind in real time. Despite this volume, it seems Costolo encourages an understanding of Twitter that remains aware of the careful handcrafted and individual nature of tweets, sent by individual *people*. This is within the larger idea of the platform's ability to function as the proverbial global town square, an image the company has been keen to emphasise. By highlighting the art installation, the space within which the tweets exist, is also rendered visible to some extent, highlighting that this is a particular kind of space, a proprietary space. Costolo thus

gives us a glimpse of 'the world', offering up a number of different vantage points, not least his own, as a significant overseer of this world. Not just an overseer, but a creator also.

This paper is concerned with considering a number of ways in which data is made, by whom and how. Rather than assuming an *a priori* existence of the data 'out there', waiting to simply be recovered and turned into insight, this paper suggests that data is co–produced and that this production is premised on multiple dynamic research intersections. Specifically, this article is concerned with how data is made by application programming interfaces (APIs), looking at Twitter in particular. It considers how researchers themselves make and select data and finally, how the tools researchers use limit the possibilities of the data that can be seen and made. Tools then in turn take on a kind of data–making agency. In all of these data making processes and encounters, data is presented, selected and thought of in particular ways, with some data apparently considered more appealing for inclusion, processing and potential analysis than others. This means that some data becomes more visible than others. This tension between visibility and invisibility is nothing new. Yet the implications within the context of Big Data are worth considering anew, given the contemporary prominence and apparent popularity of this phenomenon.

The first part of the paper briefly considers a theoretical framework within which to think about Costolo's comments concerning visibility. It then highlights two influential definitions of Big Data, the first presenting an academic perspective, the second an industry one. The article seeks to further add to these from a critical academic position, offering three new ways to define and think about Big Data. It then considers why some online data have largely escaped the attention of Big Data's all–consuming gaze. With the strong focus on text and numerical data, and the revealing of networks and connections, images and other visual material uploaded on social media are rarely considered worthy of serious interest. This is peculiar given the evidential importance and value users themselves give to them. The article then moves on to discuss the ways in which APIs, researchers and tools each make data, concluding, and making some suggestions for further work.

2. Theoretical background and definitions

It is worth exploring this idea of a 'world' being created on Twitter and in turn how it can be made visible. This section draws on theoretical ideas of how states have traditionally sought to make different types of social worlds visible and in turn, controllable. It then considers two influential definitions of Big Data, compares them and suggests further additions.

2.1. Seeing a world (of data)

Twitter is of course not 'the world', it is 'a world', but what is of interest here is the way in which Costolo encourages people to think about Twitter as a particular world along with the vantage points available for seeing it. He therefore suggests two things: first, that it is indeed possible to see everything in this world, and second, that it is possible to make sense of what is seen. It is important to remember that what you see is framed by what you are able to see or indeed want to see from within a specific ideological framework. Not everyone sees the same things in this world. Costolo's statement chimes with some of the arguments James Scott (1998) has developed in *Seeing like a state* highlighting the power relations at work within certain visually controlled built environments. Scott details a series of failed attempts and grand schemes from the twentieth century, which shared the overall aim of seeking to improve the human condition. Driven by a belief in scientific laws, the concerted efforts on the part of these states was to make the lives of those within society more legible, therefore comprehensive and ultimately controllable by state powers through the application of a rational order. Scott emphasizes the role of airplanes in enabling a 'synoptic view' [1] along with the role of architecture and city planning, highlighting the redesign of Paris into a 'spectacle' [2], in order to make it easier to see everything and everyone. Noting the potentially detrimental experiential effects on the lives of citizens, he observes: 'The fact that such order works for municipal and state authorities in administering the city is no guarantee that it works for citizens' [3]. This statement is reminiscent of the increased debates around the experiences and rights of the users of social media platforms, the ones creating the data. Here the market is adopting and embracing similar ideals about synoptic views. Not designed to control in a governmental sense *per se*, though there is clearly a regime of control in place concerning the (re)use of data. Although the projects Scott describes ultimately failed, their enduring appeal has remained and it is perhaps unsurprising that many states now take a very keen interest in data collected from social media. Visibility can be instrumentalised in different ways, depending on the interests of those seeking to make something visible. Visibility can be useful as a means of control, it can be commercially exploited, or it can be sold to others who can exploit it in turn.

In describing both the distant and close up views, Costolo highlights something else besides: what can potentially be seen at a distance, 'just black stones', and what can be observed from close up: individually crafted aesthetically pleasing little sculptures. This means that whilst one might see, or thinks one sees, certain wider patterns from a distance, the view inevitable changes when focusing on objects up close. From up close these sunflower seeds are no longer seen as a homogenous mass of black stones, but are far more complex and messier to deal with. This is similar if we think of Twitter data or any smaller sample drawn from a much larger dataset for that matter. From a distance one billion tweets may look like a fairly homogenous mass, but from close up, at the level of the individual tweet, all sorts of messiness and complexity are revealed and it becomes very difficult to account for their specific context. This is difficult to deal with analytically, but that does not mean researchers should not try.

In highlighting the art gallery setting, Twitter is also revealed as a particular kind of space, where data is proprietary and for sale. Both aesthetically pleasing artefacts, sunflower seeds and tweets, possess monetary value as well. Two years after the show, the Tate announced that it had bought eight million of the porcelain sunflower seeds for an undisclosed sum, though a small number had sold for £3.50 a seed at Sotheby's the year before (Kennedy, 2012). This monetary value of the individual seeds resonates with the increased revenue generating initiatives at Twitter, through the selling of data by approved resellers such as Gnip and DataSift.

As various researchers have pointed out, this valuing of tweets and social data more widely in monetary terms, inevitably means data can become more unreliable due to distortion that happens across metrics. Metrics get gamed because money is involved. David Karpf (2012) has suggested that there is an urgent need for the online research community to accept that the data it draws on is 'likely never going to be all that good' [4]. He summarises the problem of gamed metrics as follows: *'Any metric of digital influence that becomes financially valuable, or is used to determine newsworthiness, will become increasingly unreliable over time'* [5]. What is then important is to be aware of how this specific unreliability can be accounted for in the research process. A number of practical suggestions are developed later on in this article. But before this, it is worth reflecting on how the concept of Big Data is defined and by whom.

2.2. Defining Big Data

Because of the broad interest in Big Data from a range of different groups across academia, government and industry, it is worth considering and comparing two widely cited definitions of Big Data arising from these different arenas, the first reflecting an academic perspective, the second an industry one.

The first definition, by boyd and Crawford (2012), offered in their recent influential article on the topic, includes both cultural and technological aspects, but also highlights Big Data as a 'scholarly phenomenon' [6]. They suggest that the interplay of these three specific aspects is crucial in defining Big Data. Quoting them in full, they propose the following three–part definition, resting on the interaction between:

1. *Technology*: maximizing computation power and algorithmic accuracy to gather, analyze, link, and compare large data sets.
2. *Analysis*: drawing on large data sets to identify patterns in order to make economic, social, technical, and legal claims.
3. *Mythology*: the widespread belief that large data sets offer a higher form of intelligence and knowledge that can generate insights that were previously impossible, with the aura of truth, objectivity, and accuracy. [7]

The emphasis on myth is important in two ways. First, because it draws on making visible the ways in which myths work and what is at stake in understanding this as a process. Roland Barthes (1993) explains that the key function of a myth is to naturalize beliefs that are contingent, making them invisible, and therefore beyond question (see Jensen, this issue). It is just so. boyd and Crawford argue that we are not yet at the stage where things are 'just so', but still in motion. Things are up for grabs so to speak, before the emerging ideas about Big Data become codified and institutionalised. There is therefore an urgent imperative to question the mechanisms and assumptions around Big Data. Related to this point is what Bowker and Star (2000) highlight about the limitations of available ways in which information can be stored in society. Instead of seeing the limitations of the technical affordances and imagine different ways in which information might be structured, the ways in which information is structured become naturalized, people begin to see these structures as 'inevitable' [8]. This last insight is useful in terms of developing critical ways for making them visible, for example, how social media companies make data available through APIs. What are the other possible ways in which data could be made available, thought about and imagined?

A second important definition of Big Data comes from industry, from IT consultancy company Gartner and essentially focuses on the first two parts of the first definition, emphasising technology and analysis. The Gartner definition of Big Data is as follows: '"Big data" is high–volume, –velocity and –variety information assets that demand cost–effective, innovative forms of information processing for enhanced insight and decision making' (Gartner in Sicular, 2013). Similar to the first definition, this is a definition in three parts. The first part consisting of the three V's — volume, variety and velocity — focused on the technical infrastructures necessary to deal with vast amounts of (unstructured) data. The next two parts highlight cost–effectiveness and innovation in processing this data. The final part focuses on the other key benefits: the possibility of greater insight and thus better decision–making. Unlike the first definition, this second one offers no possibility or encouragement to reflect on 'Big Data' as a phenomenon, or to make visible inherent claims about objectivity. Kate Crawford has recently extended the work around the mythology of Big Data to highlight the problem of '"data fundamentalism," the notion that correlation always indicates causation, and that massive data sets and predictive analytics always reflect objective truth' (Crawford, 2013). This idea and belief in the existence of an objective 'truth', that something can be fully understood from a single perspective, again brings to light tensions about how the social world can be made known. Related to that: which methods and techniques (with their implied assumptions) can be used to know and understand the social world?

Returning to their definition and responding specifically to the focus of the second part of the boyd and Crawford's (2012) definition on analysis, as well as the emphasis on the three Vs in the second definition from Gartner, one might consider an alternative set of Vs from an academic perspective focused on the data–making process. They are listed below.

2.2.1. Validity

One of the key concerns within the social sciences in terms of using social media data is the anxiety around the validity and quality of the data. This takes a number of forms, both focused on the representativeness of the data. First, there is a focus on the representativeness of the data in relation to how the data is made available to the researcher. Second, there is a concern with how representative the data is in relation to a general off–line national population. So both are then concerned with how the sample is constructed and what can subsequently be read into this sample.

Again focusing on Twitter, a key concern that has recently produced some strong responses from academia, centres on better understanding how the Twitter firehose (all the publicly available tweets) relates to the various public APIs most academics have to rely on due to the cost involved in purchasing and processing firehose data. The key anxiety is not knowing exactly how Twitter samples from the firehose (although some documentation is publicly available) and this raises concerns about the overall value of this publicly available data (Gerlitz and Rieder; 2013; González–Bailón, *et al.*, 2012; Morstatter, *et al.*, 2013). This is thus a concern about making data at the level of Twitter creating the sample. Such issues around validity of the sample also arise in relation to the often opaque and unclear ways in which researchers themselves make and collect data for research purposes. The subsequent lack of sharing practices between academics in order for data to be examined by others and findings, in principle, becoming reproducible, remains an issue (Bruns, this issue).

A second key strand of concern highlights the lack of understanding of how Twitter users relate to general, national populations. In a recent study, led by Alan Mislove, *et al.* (2011), the researchers address this issue by examining over three million U.S. Twitter accounts, roughly one percent of the U.S. population and compare this to 2000 U.S. Census data. They examined the Twitter users along the axes of geography, gender, and race/ethnicity and conclude that Twitter users are significantly overrepresented in the more densely populated areas of the U.S., that users were predominantly male, and that they represented a highly non–random sample of the overall distribution of race and ethnicity. Other recent research by Eszter Hargittai and Eden Litt (2011), which highlights survey data from 505 diverse young American adults, shows that in their sample, young African Americans along with those with higher online skills are more likely to take up Twitter. They also suggest that interest in celebrity and entertainment was a significant predictor in Twitter use. In contrast, interest in news, local, national or international or politics showed no relationship to Twitter adoption in the population segment that they examined. Projects that seek to map national Twitter populations remain rare, with work led by key Twitter researchers Jean Burgess and Axel Bruns (2013), on mapping the Australian Twittersphere, a notable exception. In relation to the last study, though, it is worth introducing a final caveat. What is meant by a 'population'? Is this an online population in relation to an off–line one? So, is the question about how representative the online sample is in relation to what we know about the national population, derived for example from census data? Or is the question something else? In the case of the Burgess and Bruns study, this has a different starting point. This is about better understanding a population of Twitter users, as Twitter users. Drawing on the work of Richard Rogers (2009, 2013), this is about understanding online data as grounded in other *online data*, rather than off–line measurements. The online thus becomes the baseline. These are important fundamental differences to make explicit when we try to assess the multiple ways in which concerns arise over validity of samples, how these might be addressed, and which population they are meant to be related to exactly.

2.2.2. Venture

Thinking about the making process, 'venturing' can be a useful concept to explore and could mean a number of things: namely to offer (a take on the data), but the term can also hint at the more hazardous side of these practices. We venture into something, not presuming we already know all the answers. Thinking about venturing in this way also highlights the necessary curiosity required to want to find out about different possible worlds 'out there' expanding on what can be seen. This idea of venturing exists in contrast to frequent problematic assumptions that we can fully know the worlds we are investigating. Or, that this exploration is done from an objective position that is separate from the reality being discussed. Moreover, as Karpf (2012) highlights: 'The Internet is in a state of ongoing transformation' [9]. This then means that at best, we can venture to explore this highly dynamic and rapidly shifting world and offer partial glimpses across time. Finally, venturing is also concerned with offering a specific view when we present our findings, or a specific interpretation: We are on a mission to make a point about the data we made. This highlights the more overlooked, interpretative side of these practices, whether we deal with large or small volumes of data. As Costolo highlights in the opening anecdote, we tell stories about the data and essentially they are the stories we wish to tell.

2.2.3 Visibility

Issues around visibility have already been mentioned throughout this paper, but can be thought about in a further number of ways. Firstly, there is the visibility of the different steps taken in the process of making and dealing with data through a range of different encounters with the data. These traces often remain invisible, but can involve crucial information. Visibility also raises questions of invisibility and the tension between them already alluded to, namely what is and is not shown or is or is not seen. What is more, data is processed and turned into data visualisation: what does the data show? What does it not show and leave out? How can these visualisations be read and what kinds of skills are required for reading them? It is thus crucial to understand how visualisations are made, what they purport to show, what viewers think they show, and what they do not show. And finally, there is visibility in terms of online visual cultures, specifically in relation to the millions of images shared daily on social media, expanded on in the next section.

Following a brief provocation highlighting that not all data is of equal interest to the various research communities engaged with Big Data, the remainder of this paper is concerned with further exploring the analysis part of boyd and Crawford's (2012) definition, by looking at how data, prior to it being analysed is *made*, by whom and how, and what can be gained from understanding these processes better.

3. Provocation: Data not seen and not made

People currently produce and use a lot of images as part of their everyday lives. They do this primarily using digital media and online social media platforms and the rate at which they do this is rapidly increasing. In May 2011, 170 million tweets were sent daily, of which 1.25 percent contained a link to a picture from a photo sharing service indicating just over two million daily image shares (Levine, 2011). Similarly, two years after it was founded, in October 2010, Instagram reported that over 50 million people had shared over one billion photos through the app in October 2012 (Instagram, no date). Not one year later, in June 2013, Instagram boasted over 130 million active monthly users, 16 billion photos on the service altogether, with over one billion likes recorded *every single day* (Crook, 2013).

The use of online technologies to create and circulate images is increasing in popularity. Amongst U.K. Internet users, posting photos has significantly increased in recent years: from 44 percent of users in 2009 to 53 percent in 2011 [10]. These figures may be attributed to changes in mobile phone use in relation to which the taking and sending of photos are now the two most important activities after text messaging [11]. Moreover, within the Flickr photo–sharing community the Apple iPhone 4S is now the most popular camera used (Flickr, no date). These developments point to the increasing importance of smartphones — as all–in–one *networked* devices (Cruz and Meyer, 2012) — for the production and online circulation of personal photographs via platforms like Facebook and Twitter. The circulation of images on social media also involves the curation of 'personal digital collections' [12], often using images appropriated from elsewhere on the Internet, as indicated by the recent rise in popularity of the visual micro–blogging platform Tumblr and the image–collection platform Pinterest.

These practices involve the investment of considerable personal time and effort on the part of many people, suggesting that all sorts of cultural meaning and value are wrapped up in these activities, not least because social media allows people to show the images that they make or collect to others. This means that image related activities on social media are fundamentally communicatory and socially defined acts. As Van House and Davis (2005) point out, the presentation of personal photographs using mobile phones and social media has a number of 'social uses' that include the development and maintenance of relationships, the construction of personal and collective memory, self–presentation, and self–expression. Such social circulations of images involve collectively shared values. But more than that, they represent meeting points between personal and collective value. People send, collect, organise, and show images of people (and therefore relationships), objects, and experiences they value. They also send, collect, organise, and show already existing images that they value, often because they are valued and have meaning in a wider collective cultural context (Popescu, 2012; Zarro and Hall, 2012). Such images can be valued as forms of witnessing, as aesthetic artefacts, documents, historical records, as tokens of collective identity, and so on.

Social media companies value images differently. One could argue for example that Facebook values images in terms of their ability to attract certain types of activities by the users of the platform. This occurs most notably through enriching the image with tags, so that connections can be gleaned between users: they both appear in the same image (which may also include additional data such as location). As of January 2013 Facebook had 240 billion images stored on its service (Wilhelm, 2013). With many users tagging and enriching this data with what is likely to be reliable information (provided the tags are related to an identifiable user and also correctly identifies this user). This thus means that this large dataset provides important opportunities for machine learning and training algorithms.

Yet images are currently an under–researched area within social media research (Vis, *et al.*, 2013) as well as within the growing interest in Big Data. The images themselves do not easily lend themselves to popular Big Data 'mining' techniques and are thus typically a discarded data object in such enquiries. It therefore seems that in an academic research context at least, images are not as valued as they are by social media users themselves. The question for researchers then becomes how images can or should be valued within a research context, seen as valuable research objects within Internet research. Some suggestions about how this might be done analytically are offered in the next section. What is clear, however, is that due to the complex nature of current production, viewing (think Snapchat for example) and circulation practices, we need to identify and draw on a range of different relevant theories and methods to make sense of these emerging visual cultures on social media.

4. Practices: APIs, researchers and tools making data

Focusing now on how we can think about Big Data analysis as a practice is important. Karpf (2012) emphasises that Internet researchers should be 'question–driven, letting the nature of their inquiry determine their methods' [13]. This seems an obvious point, but in practice it is not always in evidence in current social media research. Sometimes research arises simply because data is available (for example through the donation of data), which then greatly limits the questions than can be asked because data was not originally created with the questions we now wish to ask, in mind. Data made available through APIs, including paying attention to how this is made available, can further limit the questions researchers ask. Moreover, the tools we use can limit the range of questions that might be imagined, simply because they do not fit the affordances of the tool. Not many researchers themselves have the ability or access to other researchers who

can build the required tools in line with any preferred enquiry. This then introduces serious limitations in terms of the scope of research that can be done.

4.1. APIs making data

Most social media platforms now make data publicly available on their platform through APIs. This has resulted in considerable development from third party developers to create all sorts of applications and content on top of this data. Two key companies that have emerged as key data brokers, specifically focusing on social media data from a large number of sources, are Gnip and Datasift. They have either been established recently or have refocused their attention on social media data. Gnip was founded in 2008, but since 2010 has had a strong focus on social media data. Social media platforms do not give access to 'all' the data, though. For example, data derived from enhancing activities will not necessarily be made available for sale. Added value, created for instance when a link is shortened on Twitter and turned into 'a t.co link', provides additional data, which is currently not shared. The t.co link is the standard way in which Twitter shortens links: it 'measures information such as how many times a link has been clicked, which is an important quality signal in determining how relevant and interesting each Tweet is when compared to similar Tweets' (Twitter, no date). At the time of writing, Gnip highlights that it gives full firehose access to Twitter, FourSquare and Tumblr data, while public API data is included for Facebook, Flickr, YouTube, Instagram, Vimeo, Google+ and others. This in principle means that tools could be developed that draw on this rich API data ecology and pull in data from a range of different platforms to allow for comparative analysis. While the technical data infrastructure has reached a point where this is possible, academia has yet to catch up with these possibilities. Although industry 'listening platforms' do tend to pull in data from a range of different platforms to track conversations, these tools are often unsuitable for academic purposes because of their cost, along with the problematic 'black box' nature of many of these tools.

Already extensive metadata is now regularly 'enriched' further by companies like Gnip and Datasift to produce rich opportunities for advertisers to mine it. Given the data explosion associated with social media data, it is then not surprising that an emerging industry has developed around so called 'social data', largely focused on metadata.

Gnip is keen to highlight the difference between 'social media' and 'social data'. For Gnip social media is essentially about communication and users expressing themselves, where their content is 'delivered to other users'. Social data on the other hand 'expresses social media in a computer–readable format (*e.g.*, JSON) and shares metadata about the content to help provide not only content, but context. Metadata often includes information about location, engagement and links shared. Unlike social media, social data is focused strictly on publicly shared experiences' (Ellis, 2013). As Bruhn Jensen (this issue) highlights about metadata, this data about data is a source of information in its own right, beyond what is delivered, 'sent', to other users and in turn 'received' by them. This 'meta–information' seemingly situates the content exchanged through this communication in relation to its contexts. Understanding how social media platforms make metadata along with what they make available is in constant flux. More than that, we need to better understand how the metadata offered for sale or through public APIs is often 'enriched' in ways that gloss over a variety of problems associated with this process.

For example Gnip has recently started offering something it calls 'Profile Geo enrichment' (Cairns, 2013), in order to help a 'customer understand offline locations that are relevant to online conversations' (Cairns, 2013). Location data is highly sought after within industry and academia. Gnip's company blog highlights that this new product is able to 'normalize' all the different ways in which people leave location traces online, in the first instance focusing on Twitter, stating that their customers are 'hungry to analyze Twitter through a geographic lens' (Cairns, 2013). The blog also explains how this new product relates to data available through the Twitter APIs, including the firehose. Attention is drawn to the fact that whilst only one percent of users engage in geolocation activities (resulting in fewer than two percent of all tweets containing latitude/longitude coordinates), more than half of tweets contain a location in the profile information of the user. For this reason, the profile location is thus seen as highly relevant, not least because, as Gnip argues, this offers more 'evenly distributed' information (presumably to mean 'more representative') across the Twittersphere. In other words, Gnip does not simply focus on this fraction of users that add latitude/longitude coordinates to their tweets. A product is thus created around making 15 times more geolocation data available than currently possible on Twitter, so that Gnip customers 'can now hear from the whole world of Twitter users and not just this 1 percent' (Cairns, 2013). Finally, making a leap beyond Twitter in line with Mislove, *et al.'s* (2011) research into connecting Twitter data to the general U.S. population, Gnip also emphasizes similar opportunities for grounding profile location data more widely in government–derived datasets:

> Profile location data can be used to *unlock* demographic data and other information that is not otherwise possible with activity location. For instance, U.S. Census Bureau statistics are aggregated at the locality level and can provide basic stats like household income. Profile location is also a strong indicator of activity location when one isn't provided. (Emphasis added; Cairns, 2013)

Going back to the idea of a synoptic view, what we can see here is the combination of what Costolo describes in terms of seeing the world through Twitter and, theoretically, through the techniques outlined by Scott (1998), which enabled governments to see society. In this case however, the ones seeing are commercial companies that sell the data to advertisers, keen to more accurately target specific groups of (potential) customers. Whilst the data company acknowledges the limitations of these enrichment methods, these will be hard to trace once this enhanced metadata is situated within a large database and layer on layer on metadata has been added to a user. This is especially the case when this data is also combined with additional

metadata, not derived from Twitter (so pertaining to another data infrastructure), as well as further enrichment through government statistics as outlined above. It seems evident how difficult it then becomes to assess and unpick this data once it appears in aggregate.

In early 2013 it was reported that Twitter itself was also releasing richer metadata through its various APIs. Following a series of recent restrictions within the Twitter API data ecology, self–described educational tech explorer, Martin Hawksey, highlighted in March 2013 that the Twitter Search API now gave access to considerably more metadata than before (Hawksey, 2013). Whilst there were many similar blog posts by developers at the time, including comments on the main announcement on the official Twitter developers blog, Hawskey does something very interesting: he *shows* how much more data can now be accessed through the Twitter Search API (see Figure 1). This gives an instant overview in terms of the massive expansion in data created and in turn made available through the APIs.

Figure 1: Old and new Twitter search API responses (March 2013).

Comparing the image on the left, the old API data, to the one the right, it is instantly clear that the new version of the Twitter Search API gives access to far more metadata than before. For the purposes of this paper it is worth considering whether this constitutes additional access to metadata that already existed but developers only now received access to, *or* if this additional metadata has been created, *made* by the company in order to attract more development activity around the platform. Put differently, the API itself could be seen as a data maker here. Not in the way described earlier, highlighting anxieties over the way data is sampled from the firehose, but rather as a dynamic service that constructs and gives access to new and 'enriched' metadata, pandering to the perceived desires of a seemingly insatiable data market. Writing about the politics of Twitter data, Puschmann and Burgess (2013) note the lack of consideration or visibility of the end user in these data making and trading activities: 'End–users (both private individuals and non–profit institutions) are without a place in it, except in the role of passive producers of data. The situation is likely to stay in flux, as Twitter must at once satisfy the interests of data traders and end–users, especially with regards to privacy regulation. However, as neither the contractual nor the technical regulatory instruments used by Twitter currently work in favour of end users, it is likely that they will continue to be confined to a passive role'. This statement resonates with Scott's (1998) observation about the potentially detrimental experience for citizens as part of the government structures he outlines.

4.2. Researchers making data

Researchers of course also actively make data, though this can easily be overlooked or not attributed enough consideration. How a dataset is made, including which search strategies were employed, are rarely included in research findings. Whilst there is evidently no room for lengthy discussions of this kind in a standard journal article, it is worth recording this information either in an appendix or elsewhere. Decisions about what to collect (what is in, what is out), from which API data is collected, for which period, including which metadata, including an awareness of how this collected data is itself created by APIs, are important stages in the data making process. Researchers should aim to make themselves more aware and reflect more on the process through which they have collected data and make this as transparent as possible. In line with Karpf's (2012) similar call for greater transparency, as researchers we 'should be up–front about the limitations of our data sets and research designs' [14]. Expanding on that call, it is also important to know what the limitations are in the first place. Given the descriptions of how APIs create data, this is not always a straightforward process to engage with even if the intention is to be as transparent as possible in the research process. As a community of researchers, we also have to become better at describing some of the overarching problems in *principle*, but also in *practice* seeking to explain what this actually entails, using examples where necessary.

The suggestions presented below are premised on the author's own experience of working with a mixed methods approach, where the collection of an initial Twitter dataset is typically followed by a much closer examination of a subset of this larger dataset. The focus of this small subset can be multiple, for example the examination of individual journalists and the content they have created (Vis, 2012b) or a select set of images shared during a specific event (Vis, *et al.*, 2013). Among Twitter researchers within and outside academia, it has for some time now been clear that there is need to move away from often quite simple data making strategies. In light of this, it is worth thinking about the techniques that may be useful in making better data and which disciplines may help us here.

4.2.1. More sophisticated hashtag/keyword search strategies

The research area of information retrieval (IR) offers some valuable possibilities for developing more comprehensive search strategies. In building a corpus, IR would typically treat this activity as a multi–staged process with a number of stages that build on previous ones by suggesting further informed search strategies [15].

Starting with one set of search terms, this would produce an initial set of tweets. New hashtags and keywords will be present in this data and co–location techniques could produce a new set of terms to subsequently search for. In principle you could do this until no further useful search terms are identified, though it may be difficult to assess when that point is reached. When do you know a tweet is still about your topic? This is highly context dependent and difficult to gauge at a distance, by simply seeing keyword/hashtags offered for consideration by a research tool for example. Moreover, it is possible that in some cases the same term can both be about your topic and also not be about your topic depending on the context. This then requires manual checking and domain expertise to resolve and a description of how this was resolved in the dataset.

4.2.2. Search based on links

Once these search techniques have been exhausted, more relevant data may be collected by searching again, but this time for frequently shared links [15]. These could be popular newspapers articles, blog posts, videos, or images present in the data. By applying this extra search approach one — to some extent also — gets beyond the problem of language bias that may otherwise be an issue. It is often difficult to build a strong set of candidate terms in other languages. This second technique may then identify additional data, for example tweets that use none of the hashtags or keywords, but simply share a link with a comment (for example: 'I can't believe this is happening right now': link)

Users can experience images on Twitter in one of two key ways: by accessing and viewing the image directly on the platform (if uploaded directly to Twitter) or by clicking a link that takes you to the image, for example on Instagram. At the time of writing, most are uploaded external to Twitter and shared on the platform. For researchers, it is thus challenging to deal with images that for social media users are available in accessible visual forms, but as data are effectively invisible, because they entail machine–readable hyperlinks. Making these images visible again is not easily done at scale. Researchers therefore need to find ways to navigate and gain visual access to smaller groups of images within large datasets. Consequently the analytical journey from macro–scale Big datasets to micro–scale small data interpretation is a complex one, but could fruitfully include methodological experiments in how to access images *through* the data. Once images are accessible, interpretative approaches appropriate to the specificity of images circulated through Twitter are required. These approaches will need to be sensitive to differences between images in terms of medium, genre, aesthetic form, and types of spectatorship. Traditional quantitative approaches are also not well equipped to give such insights.

4.2.3. Duplicate detection

Once the initial dataset has been made, further inspection is often needed to address two additional key issues: duplicate content and spam. Especially when collecting large volumes of data, filtering out spam content may be required. A combination of human verification and computational filtering can be very productive. What constitutes 'spam' for the purposes of the specific project may also need to be considered and is likely to be project specific.

It is also important to explain how duplicate content was dealt with. Duplicate content in the context of Twitter most notably means how retweets, modified tweets and automated retweets are identified. Manual retweets are easily identified by searching for the letters 'RT', whilst the identification of automated tweets requires an algorithmic intervention, which instructs an algorithm to compare tweets to each other and

identify identical or near identical content (for a description of this technique, see Lotan, *et al.*, 2011). What constitutes 'near identical' content will require further explanation, namely by highlighting how the algorithm operates in terms of the Levenshtein distance used to essentially say 'tweet b is still similar enough to tweet a because the modification resides within our agreed set parameters' (for a straightforward explanation, see Gilleland, no date). As tweets are such short pieces of text, a seemingly small moderation, for example the adding of '+1!' to an original tweet can quite drastically alter its meaning. It is thus imperative to explain how such potential duplicate content was dealt with in order to better understand the findings.

Finally, and this is incredibly hard to deal with at a large scale, it is important not to overlook the issue of potential problems of data including the effects of gamed metrics, including click farms, fake/spoof profiles, follower boosts, bots, and other forms of deception like swarms, link–baiting and so on, already highlighted. This then highlights an important aspect of dealing with Twitter data: it important to have domain expertise, as this will facilitate the ability to recognise limitations or identify anomalies in a given dataset. It is therefore productive to distinguish between Big Data approaches that treat Twitter data simply as a lot of data and those that approach Twitter as social media first, often as part of a media/cultural/Internet studies approach.

4.2.4. Dealing with feature changes

It is also important to be aware of the dynamic nature of the Twitter data ecosystem and platform specificities that may need to be considered as part of the data collection strategy. Twitter data collected in 2009 is difficult to compare to data collected in 2013. Features are created, updated and retired, and it is thus important to take into account the implications that may arise for making and comparing data. If such feature changes occur during the collection of data, it important to make these explicit and discuss the potential implications that may arise across the collected data.

4.3. Tools making data

Due to the analytical challenges of Big Data it is unsurprising that researchers working in this area now regularly operate as teams or research groups, and that they will now most likely include a computer scientist who can aid with the data collection and processing. It is worth noting that within such setups the making of the data is often not done by the whole team, and it thus becomes important to reflect on such strategies. What are the benefits of a team rather than a single person discussing the collection strategy and actually gathering the data? This highlights an issue about skills within such interdisciplinary teams: who is actually capable of collecting the data in the first place? Moreover, such teams will often build a set of bespoke tool solutions that others cannot use. Or, even if tools built by other computer scientists are freely available, it remains commonplace for a newly formed team to build its own tools. As a consequence of such practices tools for researching Twitter have developed in silos, and where tools are freely available it is not clear how widely these are used and what the barriers to their adoption may be. Although there have been recent calls to aim to standardise approaches within Twitter research (Bruns and Stieglitz, 2013), little attention has yet been paid to important role tools themselves play in terms of data making. Unlike industry, where different types of 'listening platforms' and tools are frequently reviewed, academia has been slow to offer similar reviews of the different tools now available for studying Twitter. Aside from the occasional discussion about the problems of the 'black box' nature of many tools, it would be incredibly helpful to critically consider what we want from our tools. What are our requirements as researchers and how can these be translated into future tool development? That is to say: what are our research questions?

Two tools created by information scientist Mike Thelwall for example, Webometric Analyst (http://lexiurl.wlv.ac.uk/) and Mozdeh (http://mozdeh.wlv.ac.uk/) are worth considering in this context. Webometric Analyst allows for the collection of social media data, including from Twitter, and offers a range of methodological approaches, such as network analysis and sentiment analysis. Mozdeh focuses on longitudinal data collection from Twitter and encourages time–series analyses, allowing the researcher to observe changes over time. The fact that these newer generations of tools move far beyond the simple archival attributes of now defunct services like TwapperKeeper is worth noting. Within their design, more critical approaches to data collection and analysis are embedded from the start. This is particularly true of Mozdeh. As tools evolve further it is important to understand and question the research limitations that are implicit in their design. Do they stop researchers asking certain types of questions? If so, what can we do about this?

Alongside a more critical understanding of the role of APIs in creating data, reflecting on our own roles as researchers in making data and describing its limitations, we also need to cast a critical eye on the tools we use. In doing this, we need to move beyond discussions about their perceived 'black box' nature, although these remain important. We need to expand our research imaginations and start first and foremost with identifying the research questions we wish to ask. Seeing through and accounting for the ideological assumptions at the heart of Big Data (boyd and Crawford, 2012; Crawford, 2013), requires more than a response that highlights the importance of showing the limitations of the questions that can be asked of Big Data. Whilst this is incredibly important work, it is also critical to have discussions about the questions we wish to ask and how we might answer them, including a careful consideration of the data, methods and tools we might need to do so. Designed with the research question as the dominant driver of the enquiry, such approaches could further develop the area of social media research in exciting new ways in the future.

About the author

Farida Vis is a Research Fellow looking at 'Big Data and Social Change', based in the Information School at the University of Sheffield, United Kingdom.

Twitter: @flygirltwo
E–mail: F [dot] vis [at] Sheffield [dot] ac [dot] uk

Acknowledgements

The author is grateful to the editors of this special issue, the reviewers and Simon Faulkner, who have all made invaluable comments on earlier drafts of this paper.

Notes

1. Scott, 1998, p. 58.

2. Scott, 1998, p. 62.

3. Scott, 1998, p. 58.

4. Karpf, 2012, p. 649.

5. Karpf, 2012, p. 650.

6. boyd and Crawford, 2012, p. 663.

7. boyd and Crawford, 2012, p. 663.

8. Bowker and Star, 2000, p. 108.

9. Karpf, 2012, p. 647.

10. Dutton and Blank, 2011, p. 29.

11. Dutton and Blank, 2011, p. 15.

12. Feinberg, *et al.*, 2012, p. 200.

13. Karpf, 2012, p. 641.

14. Karpf, 2012, p. 652.

15. I am grateful to Paul Clough and Mike Thelwall for sharing their insights with me.

16. I thank Francesco D'Orazio for extensively discussing these strategies with me recently.

References

Roland Barthes, 1993. *Mythologies*. Selected and translated by Annette Lavers. London: Vintage.

Geoffrey C. Bowker and Susan Leigh Star, 2000. Sorting things out: Classification and its consequences. Cambridge, Mass.: MIT Press.

danah boyd and Kate Crawford, 2012. "Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon," *Information, Communication & Society*, volume 15, number 5, pp. 662–679.
doi: http://dx.doi.org/10.1080/1369118X.2012.678878, accessed 24 September 2013.

Axel Bruns and Stefan Stieglitz, 2013. "Towards more systematic Twitter analysis: Metrics for tweeting activities," *International Journal of Social Research Methodology*, volume 16, number 2, pp. 91–108.
doi: http://dx.doi.org/10.1080/13645579.2012.756095, accessed 24 September 2013.

Jean Burgess and Axel Bruns, 2013. "Mapping the Australian Twittershere," paper presented at Media in Transition 8 Conference (3–5 May, Boston); slides at http://www.slideshare.net/Snurb/mit8-burgessbruns, accessed 24 September 2013.

Ian Cairns, 2013. "Get more geodata From Gnip with our new profile geo enrichment," Gnip Company Blog (22 August), at http://blog.gnip.com/tag/geolocation/, accessed 13 September 2013.

Kate Crawford, 2013. "The hidden biases in Big Data," *Harvard Business Review (HBR) Blog Network* (1 April), at http://blogs.hbr.org/2013/04/the-hidden-biases-in-big-data/, accessed 10 September 2013.

Jordan Crook, 2013. "Instagram crosses 130 Million Users, with 16 billion photos and over 1 billion likes per day," *TechCrunch* (20 June), at http://techcrunch.com/2013/06/20/instagram-crosses-130-million-users-with-16-billion-photos-and-over-1-billion-likes-per-day/, accessed 4 September 2013.

Elaine Ellis, 2013. "Social data vs social media," *Gnip Company Blog* (3 May), at http://blog.gnip.com/social-data-vs-social-media-2/, accessed 10 September 2013.

Edgar Gómez Cruz and Eric T. Meyer, 2012. "Creation and control in the photographic process: iPhones and the emerging fifth moment of photography," *Photographies*, volume 5, number 2, pp. 203–221. doi: http://dx.doi.org/10.1080/17540763.2012.702123, accessed 24 September 2013.

William H. Dutton and Grant Blank, 2011. "Next generation users: The Internet in Britain," *Oxford Internet Survey 2011 Report*, at http://www.worldinternetproject.net/_files/_Published/23/820_oxis2011_report.pdf, accessed 18 August 2013.

Melanie Feinberg, Gary Geisler, Eryn Whitworth, and Emily Clark, 2012. "Understanding personal digital collections: An interdisciplinary exploration," *DIS '12: Proceedings of the Designing Interactive Systems Conference*, pp. 200–209. doi: http://dx.doi.org/10.1145/2317956.2317988, accessed 24 September 2013.

Flickr, no date. "Camera finder," at http://www.flickr.com/cameras, accessed 15 April 2013.

Carolin Gerlitz and Bernhard Rieder, 2013. "Mining one percent of Twitter: Collections, baselines, sampling,&edquo; *M/C Journal*, volume 16, number 2, at http://journal.media-culture.org.au/index.php/mcjournal/article/viewArticle/620, accessed 14 September 2013.

Michael Gilleland, no date. "Levenshtein distance, in three flavors," at http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm, accessed 10 September 2013.

Sandra González–Bailón, Ning Wang, Alejandro Rivero, Javier Borge–Holthoefer, and Jamir Moreno, 2012. "Assessing the bias in communication networks samples from Twitter," at http://arxiv.org/ftp/arxiv/papers/1212/1212.1684.pdf, accessed 14 September 2013.

Eszter Hargittai and Eden Litt, 2011. "The tweet smell of celebrity success: Explaining variation in Twitter adoption among a diverse group of young adults," *New Media & Society*, volume 13, number 5, pp. 824–842. doi: http://dx.doi.org/10.1177/1461444811405805, accessed 24 September 2013.

Martin Hawksey, 2013. "Twitter throws a bone: Increased hits and metadata in Twitter Search API 1.1" (28 March), at http://mashe.hawksey.info/2013/03/twitter-throws-a-bone-increased-hits-and-metadata-in-twitter-search-api-1-1/, accessed 10 September 2013.

Instagram, no date. "2 years later: The first Instagram photo," at http://blog.instagram.com/post/27359237977/2-years-later-the-first-instagram-photo, accessed 4 September 2013.

David Karpf, 2012. "Social science research methods in Internet time," *Information, Communication & Society*, volume 15, number 5, pp. 636–661. doi: http://dx.doi.org/10.1080/1369118X.2012.665468, accessed 24 September 2013.

Maev Kennedy, 2012. "Tate buys eight million Ai Weiwei sunflower seeds," *Guardian* (5 March), at http://www.theguardian.com/artanddesign/2012/mar/05/tate-ai-weiwei-sunflower-seeds, accessed 14 September 2013.

Sheldon Levine, 2011. "How people currently share pictures on Twitter," *Sysomos Blog* (2 June), at http://blog.sysomos.com/2011/06/02/, accessed 14 September 2013.

Gilad Lotan, Erhardt Graeff, Mike Ananny, Devin Gaffney, Ian Pearce, and danah boyd, 2011. "The revolutions were tweeted: Information flows During the 2011 Tunisian and Egyptian revolutions," *International Journal of Communication*, volume 5, at http://ijoc.org/index.php/ijoc/article/view/1246, accessed 24 September 2013.

Alan Mislove, Sune Lehman, Yong–Yeol Ahn, Jukka–Pekka Onnela and J. Niels Rosenquist, 2011. "Understanding the demographics of Twitter users," *Proceedings of the Fifth International AAAI Conference on Weblogs and Social Media*, pp. 554–557, at https://www.aaai.org/ocs/index.php/ICWSM/ICWSM11/paper/view/2816, accessed 14 September 2013.

Fred Morstatter, Jürgen Pfeffer, Huan Liu, and Kathleen M. Carley, 2013. "Is the sample good enough? Comparing data from Twitter's streaming API with Twitter's firehose," *Association for the Advancement of Artificial Intelligence Conference*, at http://www.public.asu.edu/~fmorstat/paperpdfs/icwsm2013.pdf, accessed 14 September 2013.

Richard Rogers, 2013. *Digital methods*. Cambridge, Mass.: MIT Press.

Richard Rogers, 2009. *The end of the virtual: Digital methods*. *Oratiereeks/University of Amsterdam, Faculty of Humanities*, number 339. Amsterdam: Vossiuspers UvA.

Ana–Maria Popescu, 2012. "Pinteresting: Towards a better understanding of user interests," *DUBMMSM '12: Proceedings of the 2012 Workshop on Data–Driven User Behavioral Modelling and Mining From Social Media*, pp. 11–12. doi: http://dx.doi.org/10.1145/2390131.2390136, accessed 24 September 2013.

Cornelius Puschmann and Jean Burgess, 2013. "The politics of Twitter data," In: Katrin Weller, Axel Bruns, Jean Burgess, Merja Mahrt, and Cornelius Puschmann (editors). *Twitter and society*. New York: Peter Lang.

John C. Scott, 1998. *Seeing like a state: How certain schemes to improve the human condition have failed*. New Haven, Conn.: Yale University Press.

Svetlana Sicular, 2013. "Gartner's Big Data definition consists of three parts, not to be confused with three 'V's," *Forbes* (27 March), at http://www.forbes.com/sites/gartnergroup/2013/03/27/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/, accessed 18 August 2013.

Twitter, no date. "About Twitter's link service (http://t.co)," *Twitter Help Center*, at https://support.twitter.com/articles/109623-about-twitter-s-link-service-http-t-co, accessed 15 September 2013.

Nancy A. Van House and Marc Davis, 2005. "The social life of cameraphone images," *Proceedings of the Pervasive Image Capture and Sharing: New Social Practices and Implications for Technology Workshop (PICS 2005) at the Seventh International Conference on Ubiquitous Computing (UbiComp 2005)*, at http://people.ischool.berkeley.edu/~vanhouse/Van%20House,%20Davis%20-%20The%20Social%20Life%20of%20Cameraphone%20Images.pdf, accessed 14 September 2013.

Farida Vis, 2012a. "'Twitter brings you closer': The importance of seeing the little data in Big Data," In: Drew Hemment and Charlie Gere (editors). *FutureEverybody: FutureEverything Report*, pp. 43–45, at http://futureeverything.org/FutureEverybody.pdf, accessed 10 September 2013.

Farida Vis, 2012b. "Twitter as a reporting tool for breaking news: Journalists tweeting the 2011 UK riots," *Digital Journalism*, volume 1, number 1, pp. 27–47.
doi: http://dx.doi.org/10.1080/21670811.2012.741316, accessed 24 September 2013.

Farida Vis, Simon Faulkner, Katy Parry, Yana Manykhina, and Lisa Evans, 2013. "Twitpic–ing the riots: Analysing images shared on Twitter during the 2011 UK riots," In: Katrin Weller, Axel Bruns, Jean Burgess, Merja Mahrt, and Cornelius Puschmann (editors). *Twitter and society*. New York: Peter Lang.

Todd Wasserman, 2012. "Costolo on Twitter's purpose: 'We bring people closer'," *Mashable* (18 September), at http://mashable.com/2012/09/18/costolo-today-show/, accessed 18 August 2013.

Alex Wilhelm, 2013. "Facebook: Our 1 billion users have uploaded 240 billion photos, made 1 trillion connections," *TNW* (15 January), at http://thenextweb.com/facebook/2013/01/15/facebook-our-1-billion-users-have-uploaded-240-billion-photos-made-1-trillion-connections/, accessed 13 September 2013.

Michael Zarro and Catherine Hall, 2012. "Pinterest: Social collecting for #linking #using #sharing," *JCDL '12: Proceedings of the 12th ACM/IEEE–CS Joint Conference on Digital Libraries*, pp. 417–418.
doi: http://dx.doi.org/10.1145/2232817.2232919, accessed 24 September 2013.

Editorial history

Received 16 September 2013; accepted 17 September 2013.

A critical reflection on Big Data: Considering APIs, researchers and tools as data makers
by Farida Vis.
First Monday, Volume 18, Number 10 - 7 October 2013
https://journals.uic.edu/ojs/index.php/fm/rt/printerFriendly/4878/3755
doi:10.5210/fm.v18i10.4878.

Reading 3

Creating APIs with Python and Flask

Creating Web APIs with Python and Flask

2017-08-01

Contents

Lesson Goals

Web APIs are tools for making information and application functionality accessible over the internet. In this lesson, you will:

- Learn what an API is and when you should use one.
- Learn how to build a web API that returns data to its users.
- Learn some principles of good API design, applying them to an API that draws book metadata from a database.

Setting Up

Prerequisites

You can use the Windows, OSX, or Linux operating systems to complete this tutorial, and those few instructions that are not the same across platforms will be explicitly noted. Python 3, the Flask web framework, and a web browser are required for this tutorial, and installation instructions for all platforms are outlined below.

The only knowledge explicitly assumed for this lesson is the ability to use a text editor, such as BBEdit on OSX or Notepad++ on Windows. However, knowledge of the command line, Python, and web concepts such as HTTP may make this tutorial easier to follow. If you're new to Python, consider working through the Programming Historian series on dealing with online sources to familiarize yourself with fundamental concepts in Python programming.

Installing Python and Flask

For this tutorial, you will need Python 3 and the Flask web framework. You'll also require a web browser (such as Firefox) and a text editor (such as Notepad++ or BBEdit).

To download Python, follow this link, select the button that says `Download Python 3.x.x`, and then run the installer as you normally would to install applications on your operating system. The default settings should be fine.

To confirm that Python installed successfully, first open the command line. In OSX, click the finder on the top left of your desktop (the magnifying glass) and type `terminal`. The terminal should be the first application that appears. On Windows, click the Start menu icon and type `cmd` in the search box, then press `Enter`.

Once your command line is open, enter these commands:

```
python --version
pip --version
```

If the output for these commands includes a version number, Python is installed and available from the command line and you can proceed to the next step.

Next, you'll need to install Flask. At the command line, type

```
pip install flask
```

This will install Flask using the pip package manager for Python. You should see some output ending in a notification that Flask has been installed successfully.

As an alternative to the above installation instructions, you can install the Python 3 version of Anaconda, which can be downloaded here. Anaconda comes with Flask, so if you go this route you will not need to install Flask using the pip package manager.

If you're running into trouble installing Python, you may find this Programming Historian article on installing Python helpful. Note that the instructions in that tutorial are for installing Python 2—make sure you choose Python 3 when downloading installers from the Python website, since this tutorial uses Python 3.

If you don't have a preferred text editor, I recommend BBEdit for OSX or Notepad++ for Windows.

Introducing APIs

What is an API?

If you've heard the term API before, chances are it's been used not to refer to APIs in general, but instead to a specific kind of API, the web API. A web API allows for information or functionality to be manipulated by other programs via the internet. For example, with Twitter's web API, you can write a program in a language like Python or Javascript that can perform tasks such as favoriting tweets or collecting tweet metadata.

In programming more generally, the term API, short for Application Programming Interface, refers to a part of a computer program designed to be used or manipulated by another program, as opposed to an interface designed to be used or manipulated by a human. Computer programs frequently need to communicate amongst themselves or with the underlying operating system, and APIs are one way they do it. In this tutorial, however, we'll be using the term API to refer specifically to web APIs.

When to Create an API

In general, consider an API if:

1. Your data set is large, making download via FTP unwieldy or resource-intensive.
2. Your users will need to access your data in real time, such as for display on another website or as part of an application.
3. Your data changes or is updated frequently.
4. Your users only need access to a part of the data at any one time.

5. Your users will need to perform actions other than retrieve data, such as contributing, updating, or deleting data.

If you have data you wish to share with the world, an API is one way you can get it into the hands of others. However, APIs are not always the best way of sharing data with users. If the size of the data you are providing is relatively small, you can instead provide a "data dump" in the form of a downloadable JSON, XML, CSV, or SQLite file. Depending on your resources, this approach can be viable up to a download size of a few gigabytes.

Remember that you can provide both a data dump and an API, and individual users may find one or the other to better match their use case. Open Library, for example, provides both a data dump and an API, each of which serves different use cases for different users.

API Terminology

When using or building APIs, you will encounter these terms frequently:

- **HTTP (Hypertext Transfer Protocol)** is the primary means of communicating data on the web. HTTP implements a number of "methods," which tell which direction data is moving and what should happen to it. The two most common are GET, which pulls data from a server, and POST, which pushes new data to a server.
- **URL (Uniform Resource Locator)** - An address for a resource on the web, such as `https://programminghistorian.org/about`. A URL consists of a **protocol** (`http://`), domain (`programminghistorian.org`), and optional **path** (`/about`). A URL describes the location of a specific resource, such as a web page. When reading about APIs, you may see the terms `URL`, `request`, `URI`, or `endpoint` used to describe adjacent ideas. This tutorial will prefer the terms URL and request to avoid complication. You can follow a URL or make a GET request in your browser, so you won't need any special software to make requests in this tutorial.
- **JSON (JavaScript Object Notation)** is a text-based data storage format that is designed to be easy to read for both humans and machines.

JSON is generally the most common format for returning data through an API, XML being the second most common.

- **REST (REpresentational State Transfer)** is a philosophy that describes some best practices for implementing APIs. APIs designed with some or all of these principles in mind are called REST APIs. While the API outlined in this lesson uses some REST principles, there is a great deal of disagreement around this term. For this reason, I do not describe the example APIs here as REST APIs, but instead as web or HTTP APIs.

Using APIs

Why Use APIs as a Researcher?

The primary focus of this lesson is on creating an API, not exploring or using an API that has already been implemented. However, before we start building our own API, it may be useful to discuss how APIs are useful for researchers. In this section, we'll see how APIs can be useful for approaching historical, textual, or sociological questions using a "macroscopic" or "distant reading" approach that makes use of relatively large amounts of information. In doing so, we'll familiarize ourselves with the basic elements of a good API. Considering APIs from the perspective of a user will come in useful when we begin to design our own API later in the lesson.

AN API Case Study: Sensationalism and Historical Fires

Imagine that our research area is sensationalism and the press: has newspaper coverage of major events in the United States become more or less sensational over time? Narrowing the topic, we might ask whether press coverage of, for example, urban fires has increased or decreased with government reporting on fire-related relief spending.

While we won't be able to explore this question thoroughly, we can begin to approach this research space by collecting historical data on newspaper coverage of fires using an API—in this case, the Chronicling America Historical Newspaper API. The Chronicling America API allows access to metadata and text for millions of scanned newspaper pages. In addition, unlike many other APIs, it also does not require an authentication process, allowing us to immediately explore the available data without signing up for an account.

Our initial goal in approaching this research question is to find all newspaper stories in the Chronicling America database that use the term "fire." Typically, use of an API starts with its documentation. On the Chronicling America API page, we find two pieces of information critical for getting the data we want from

the API: the API's **base URL** and the **path** corresponding to the function we want to perform on the API—in this case, searching the database.

Our base URL is:

`http://chroniclingamerica.loc.gov`

All requests we make to the API must begin with this portion of the URL. All APIs have a base URL like this one that is the same across all requests to the API.

Our path is:

`/search/pages/results/`

If we combine the base URL and the path together into one URL, we'll have created a request to the Chronicling America API that returns all available data in the database:

`http://chroniclingamerica.loc.gov/search/pages/results/`

If you visit the link above, you'll see all items available in Chronicling America (12,243,633 at the time of writing), , not just the entries related to our search term, "fire." This request also returns a formatted HTML view, rather than the structured view we want to use to collect data.

According to the Chronicling America documentation, in order to get structured data specifically relating to fire, we need to pass one more kind of data in our request: **query parameters**.

`http://chroniclingamerica.loc.gov/search/pages/results/?format=json&proxtext=fire`

The query parameters follow the `?` in the request, and are seperated from one another by the `&` symbol. The first query parameter, `format=json`, changes the returned data from HTML to JSON. The second, `proxtext=fire`, narrows the returned entries to those that include our search term.

If you follow the above link in your browser, you'll see a structured list of the items in the database related to the search term "fire." The format of the returned data is called JSON, and is a structured format that looks like this excerpt from the Chronicling America results:

```
"city": [
          "Washington"
      ],
      "date": "19220730",
      "title": "The Washington Herald.",
      "end_year": 1939,
```

By making requests to the Chronicling America API, we've accessed information on news stories that contain the search term "fire," and returned data that includes the date of publication and the page the article appears on. If we were to pursue this research question further, a next step might be finding how many

stories relating to fire appear on a newspaper's front page over time, or perhaps cleaning the data to reduce the number of false positives. As we have seen, however, exploring an API can be a useful first step in gathering data to tackle a research question.

Note that in this section, we skipped an important step: finding an appropriate API in the first place. Some resources for researching APIs are available at the end of this lesson.

What Users Want in an API

As we've learned, documentation is a user's starting place when working with a new API, and well-designed URLs make it easier for users to intuitively find resources. Because they help users to quickly access information through your API, these elements—documentation and well-conceived URLs—are the *sine qua non* of a good API. We'll discuss these elements in greater depth later in this tutorial.

As you use other APIs in your research, you'll develop a sense of what makes a good API from the perspective of a potential user. Just as strong readers often make strong writers, using APIs created by others and critically evaluating their implementation and documentation will help you better design your own APIs.

Implementing Our API

Overview

This section will show you how to build a prototype API using Python and the Flask web framework. Our example API will take the form of a distant reading archive—a book catalog that goes beyond standard bibliographic information to include data of interest to those working on digital projects. In this case, besides title and date of publication, our API will also serve the first sentence of each book. This should be enough data to allow us to envision some potential research questions without overwhelming us as we focus on the design of our API.

We'll begin by using Flask to create a home page for our site. In this step, we'll learn the basics of how Flask works and make sure our software is configured correctly. Once we have a small Flask application working in the form of a home page, we'll iterate on this site, turning it into a functioning API.

Creating a Basic Flask Application

Flask is a web framework for Python, meaning that it provides functionality for

building web applications, including managing HTTP requests and rendering templates. In this section, we will create a basic Flask application. In later sections, we'll add to this application to create our API. Don't worry if you don't understand each individual line of code yet—explanations will be forthcoming once you have this initial version of the application working.

Why Flask?

Python has a number of web frameworks that can be used to create web apps and APIs. The most well-known is Django, a framework that has a set project structure and which includes many built-in tools. This can save time and effort for experienced programmers, but can be overwhelming. Flask applications tend to be written on a blank canvas, so to speak, and so are more suited to a contained application such as our prototype API.

First, create a new folder on your computer that will serve as a project folder. This can be in your `Desktop` folder, but I recommend creating a dedicated `projects` folder for this and similar projects. This tutorial will assume that the files related to this lesson will be stored in a folder called `api` inside a folder named `projects` in your home directory. If you need help with navigation on the command line, see the Programming Historian Introduction to the Bash Command Line for the OSX and Linux command line or the Introduction to the Windows Command Line with PowerShell for Windows.

In OSX, you can directly create a an `api` folder inside a `projects` folder in your home directory with this terminal command:

```
mkdir -p ~/projects/api
```

On Windows, you can create the `api` folder with these commands in your `cmd` command line environment:

```
md projects
cd projects
md api
```

You can also create the `projects` and `api` folders using your operating system's graphical user interface.

Next, open a text editor (such as Notepad++ or BBEdit) and enter the following code:

```
import flask

app = flask.Flask(__name__)
app.config["DEBUG"] = True


@app.route('/', methods=['GET'])
def home():
    return "<h1>Distant Reading Archive</h1><p>This site is a prototype API for distant read
```

```
app.run()
```

Save this code as `api.py` in the `api` folder you created for this tutorial.

Running the Application

In the command line, navigate to your `api` folder:

```
cd projects/api
```

You can check if you're in the correct folder by running the `pwd` command. Once you're in your project directory, run the Flask application with the command:

```
python api.py
```

You should see output similar to this:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

You may also see some lines related to debugging. This message means that Flask is running your application locally (on your computer) at that address. Follow the link above, http://127.0.0.1:5000/, using your web browser to see the running application:

{% include figure.html filename="welcome.png" caption="The home page when rendered in a browser." %}

Congratulations, you've created a working web application!

What Flask Does

Now that we have a homepage for our archive, let's talk about how Flask works and what the above code is doing.

Flask maps HTTP requests to Python functions. In this case, we've mapped one URL path ('`/`') to one function, `home`. When we connect to the Flask server at http://127.0.0.1:5000/, Flask checks if there is a match between the path provided and a defined function. Since `/`, or no additional provided path, has been mapped to the `home` function, Flask runs the code in the function and displays the returned result in the browser. In this case, the returned result is HTML markup for a home page welcoming visitors to the site hosting our API.

The process of mapping URLs to functions is called **routing**. The

```
@app.route('/', methods=['GET'])
```

syntax is the part of the program that lets Flask know that this function, `home`, should be mapped to the path `/`. The `methods` list (`methods=['GET']`) is a keyword argument that lets Flask know what kind of HTTP requests are allowed.

We'll only be using `GET` requests in this tutorial, but many web applications need to use both `GET` requests (to send data) and `POST` requests (to receive data).

Below are brief explanations of the other components of the application:

`import flask` — Imports the Flask library, making the code available to the rest of the application.

`app = flask.Flask(__name__)` — Creates the Flask application object, which contains data about the application and also methods (object functions) that tell the application to do certain actions. The last line, `app.run()`, is one such method.

`app.config["DEBUG"] = True` — Starts the debugger. With this line, if your code is malformed, you'll see an error when you visit your app. Otherwise you'll only see a generic message such as `Bad Gateway` in the browser when there's a problem with your code.

`app.run()` — A method that runs the application server.

While it's useful to have a familiarity with what's going on in the script, don't worry if you don't understand precisely what every element is doing at this stage. If you understand the general outline of how this portion works, the details of how Flask renders pages are likely to become more understandable as we continue to develop our API.

Creating the API

Now that we have a running Flask application and know a little about what Flask does, we're finally ready to implement a small API with data that we'll define right in our application.

We'll be adding our data as a list of Python dictionaries. Dictionaries in Python group pairs of keys and values, like this:

```
{
    'key': 'value',
    'key': 'value'
}
```

The key identifies the type of information represented, such as `title` or `id`. The value is the actual data. For example, a short telephone book might take this format:

```
[
    {
        'name': 'Alexander Graham Bell',
        'number': '1-333-444-5555'
    },
    {
```

```
        'name': 'Thomas A. Watson',
        'number': '1-444-555-6666'
    }
]
```

The above phone book is a list of two dictionaries. Each dictionary is a phone book entry consisting of two keys, `name` and `number`, each paired with a value that provides the actual information.

Let's add some data—entries on three science fiction novels—as a list of dictionaries. Each dictionary will contain ID number, title, author, first sentence, and year of publication for each book. Finally, we'll add a new function: a route that will allow a visitor to access our data.

Replace our previous code in `api.py` with the code below:

```
import flask
from flask import request, jsonify

app = flask.Flask(__name__)
app.config["DEBUG"] = True

# Create some test data for our catalog in the form of a list of dictionaries.
books = [
    {'id': 0,
     'title': 'A Fire Upon the Deep',
     'author': 'Vernor Vinge',
     'first_sentence': 'The coldsleep itself was dreamless.',
     'year_published': '1992'},
    {'id': 1,
     'title': 'The Ones Who Walk Away From Omelas',
     'author': 'Ursula K. Le Guin',
     'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival c
     'published': '1973'},
    {'id': 2,
     'title': 'Dhalgren',
     'author': 'Samuel R. Delany',
     'first_sentence': 'to wound the autumnal city.',
     'published': '1975'}
]


@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''
```

```
# A route to return all of the available entries in our catalog.
@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)

app.run()
```

Run the code (navigate to your `api` folder in the command line and enter `python api.py`). Once the server is running, visit our route URL to view the data in the catalog:

http://127.0.0.1:5000/api/v1/resources/books/all

You should see JSON output for the three entries in our test catalog. Flask provides us with a `jsonify` function that allows us to convert lists and dictionaries to JSON format. In the route we created, our book entries are converted from a list of Python dictionaries to JSON before being returned to a user.

At this point, you've created a working, if limited, API. In the next section, we'll allow users to find books via more specific data, such as an entry's ID.

Finding Specific Resources

Right now, users can only view our entire database—they can't filter or find specific resources. While this isn't a problem with our test catalog, this will quickly become less useful as we add data. In this section, we'll add a function that allows users to filter their returned results using a more specific request.

Below is the code for our new application with filtering capability. As before, we'll examine the code more carefully once you have it running.

```
import flask
from flask import request, jsonify

app = flask.Flask(__name__)
app.config["DEBUG"] = True

# Create some test data for our catalog in the form of a list of dictionaries.
books = [
    {'id': 0,
     'title': 'A Fire Upon the Deep',
     'author': 'Vernor Vinge',
     'first_sentence': 'The coldsleep itself was dreamless.',
     'year_published': '1992'},
    {'id': 1,
     'title': 'The Ones Who Walk Away From Omelas',
     'author': 'Ursula K. Le Guin',
     'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival o
```

```
     'published': '1973'},
    {'id': 2,
     'title': 'Dhalgren',
     'author': 'Samuel R. Delany',
     'first_sentence': 'to wound the autumnal city.',
     'published': '1975'}
]


@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''


@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)


@app.route('/api/v1/resources/books', methods=['GET'])
def api_id():
    # Check if an ID was provided as part of the URL.
    # If ID is provided, assign it to a variable.
    # If no ID is provided, display an error in the browser.
    if 'id' in request.args:
        id = int(request.args['id'])
    else:
        return "Error: No id field provided. Please specify an id."

    # Create an empty list for our results
    results = []

    # Loop through the data and match results that fit the requested ID.
    # IDs are unique, but other fields might return many results
    for book in books:
        if book['id'] == id:
            results.append(book)

    # Use the jsonify function from Flask to convert our list of
    # Python dictionaries to the JSON format.
    return jsonify(results)

app.run()
```

Once you've updated your API with the `api_id` function, run your code as

before (`python api.py` from your `api` directory) and visit the below URLs to test the new filtering capability:

127.0.0.1:5000/api/v1/resources/books?id=0
127.0.0.1:5000/api/v1/resources/books?id=1
127.0.0.1:5000/api/v1/resources/books?id=2
127.0.0.1:5000/api/v1/resources/books?id=3

Each of these should return a different entry, except for the last, which should return an empty list: `[]`, since there is no book for which the id value is 3. (Counting in programming typically starts from 0, so id=3 would be a request for a nonexistent fourth item.) In the next section, we'll explore our updated API in more detail.

Understanding Our Updated API

In this code, we first create a new function, called `api_root`, with the `@app.route` syntax that maps the function to the path '/api/v1/resources/books". That means that this function will run when we access http://127.0.0.1:5000/api/v1/resources/books.

Inside our function, we do two things:

First, examine the provided URL for an id and select the books that match that id. The id must be provided like this: `?id=0`. Data passed through URLs like this (after the `?`) are called **query parameters**—we've seen them before when we worked with the Chronicling America API. They're a feature of HTTP used for filtering for specific kinds of data.

This part of the code determines if there is a query parameter, like `?id=0`, and then assigns the provided ID to a variable.

```
if 'id' in request.args:
    id = int(request.args['id'])
else:
    return "Error: No id field provided. Please specify an id."
```

Then this section moves through our test catalog of books, matches those books that have the provided ID, and appends them to the list that will be returned to the user:

```
for book in books:
    if book['id'] == id:
        results.append(book)
```

Finally, the `return jsonify(results)` line takes the list of results and renders them in the browser as JSON.

If you've gotten this far, you've created an actual API. Celebrate! At the end of this lesson, you'll be exposed to a somewhat more complex API that uses a

database, but most of the principles and patterns we've used so far will still apply. In the next section, we'll discuss some guidelines for creating a well-designed API that others will actually want to use. In the last section of the tutorial, we'll apply these principles to a version of our API that pulls in results from a database.

API Design Principles

Thus far, we've created a working API with test data that we've provided right in our application. Our next version of our API will pull in data from a database before providing it to a user. It will also take additional query parameters, allowing users to filter by fields other than ID.

Before building more functionality into our application, let's reflect on some of the API design decisions that we've made so far. Two aspects of a good API are usability and maintainability, and as we build more functionality into our API, we'll be keeping many of the following considerations in mind.

Designing Requests

The prevailing design philosophy of modern APIs is called REST. For our purposes, the most important thing about REST is that it's based on the four methods defined by the HTTP protocol: POST, GET, PUT, and DELETE. These correspond to the four traditional actions performed on data in a database: CREATE, READ, UPDATE, and DELETE. In this tutorial, we'll only be concerned with GET requests, which correspond to reading from a database.

Because HTTP requests are so integral to using a REST API, many design principles revolve around how requests should be formatted. We've already created one HTTP request, which returns all books provided in our sample data. To understand the considerations that go into formatting this request, let's first consider a weak or poorly-designed example of an API endpoint:

`http://api.example.com/getbook/10`

The formatting of this request has a number of issues. The first is semantic—in a REST API, our verbs are typically `GET`, `POST`, `PUT`, or `DELETE`, and are determined by the request method rather than in the request URL. That means that the word "get" should not appear in our request, since "get" is implied by the fact that we're using a HTTP GET method. In addition, resource collections such as `books` or `users` should be denoted with plural nouns. This makes it clear when an API is referring to a collection (`books`) or an entry (`book`). Incorporating these principles, our API would look like this:

`http://api.example.com/books/10`

The above request uses part of the path (`/10`) to provide the ID. While this is not an uncommon approach, it's somewhat inflexible—with URLs constructed in this manner, you can generally only filter by one field at a time. Query parameters allow for filtering by multiple database fields and make more sense when providing "optional" data, such as an output format:

```
http://api.example.com/books?author=Ursula+K.+Le Guin&published=1969&output=xml
```

When designing how requests to your API should be structured, it also makes sense to plan for future additions. Even if the current version of your API serves information on only one type of resource—`books`, for example—it makes sense to plan as if you might add other resources or non-resource functionality to your API in the future:

```
http://api.example.com/resources/books?id=10
```

Adding an extra segment on your path such as "resources" or "entries" gives you the option to allow users to search across all resources available, making it easier for you to later support requests such as these:

```
https://api.example.com/v1/resources/images?id=10
https://api.example.com/v1/resources/all
```

Another way to plan for your API's future is to add a version number to the path. This means that, should you have to redesign your API, you can continue to support the old version of the API under the old version number while releasing, for example, a second version (`v2`) with improved or different functionality. This way, applications and scripts built using the old version of your API won't cease to function after your upgrade.

After incorporating these design improvements, a request to our API might look like this:

```
https://api.example.com/v1/resources/books?id=10
```

Documentation and Examples

Without documentation, even the best-designed API will be unusable. Your API should have documentation describing the resources or functionality available through your API that also provides concrete working examples of request URLs or code for your API. You should have a section for each resource that describes which fields, such as `id` or `title`, it accepts. Each section should have an example in the form of a sample HTTP request or block of code.

A fairly common practice in documenting APIs is to provide annotations in your code that are then automatically collated into documentation using a tool such as Doxygen or Sphinx. These tools create documentation from **docstrings**—comments you make on your function definitions. While this kind of documentation is a good idea, you shouldn't consider your job done if you've

only documented your API to this level. Instead, try to imagine yourself as a potential user of your API and provide working examples. In an ideal world, you would have three kinds of documentation for your API: a reference that details each route and its behavior, a guide that explains the reference in prose, and at least one or two tutorials that explain every step in detail.

For inspiration on how to approach API documentation, see the New York Public Library Digital Collections API, which sets a standard of documentation achievable for many academic projects. For an extensively documented (though sometimes overwhelming) API, see the MediaWiki Action API, which provides documentation to users who pass partial queries to the API. (In our example above, we returned an error on a partial query.) For professionally maintained API documentation examples, consider the various New York Times APIs or the Marvel API.

Connecting Our API to a Database

This last example of our Distant Reading Archive API pulls in data from a database, implements error handling, and can filter books by publication date. The database used is SQLite, a lightweight database engine that is supported in Python by default. SQLite files typically end with the `.db` file extension.

Before we modify our code, first download the example database from this location and copy the file to your `api` folder using your graphical user interface. The final version of our API will query this database when returning results to users.

Copy the below code into your text editor. As before, we'll examine the code more closely once you have it running.

```
import flask
from flask import request, jsonify
import sqlite3

app = flask.Flask(__name__)
app.config["DEBUG"] = True

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d


@app.route('/', methods=['GET'])
def home():
```

```
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''


@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM books;').fetchall()

    return jsonify(all_books)



@app.errorhandler(404)
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not be found.</p>", 404


@app.route('/api/v1/resources/books', methods=['GET'])
def api_filter():
    query_parameters = request.args

    id = query_parameters.get('id')
    published = query_parameters.get('published')
    author = query_parameters.get('author')

    query = "SELECT * FROM books WHERE"
    to_filter = []

    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
    if author:
        query += ' author=? AND'
        to_filter.append(author)
    if not (id or published or author):
        return page_not_found(404)

    query = query[:-4] + ';'

    conn = sqlite3.connect('books.db')
```

```
    conn.row_factory = dict_factory
    cur = conn.cursor()

    results = cur.execute(query, to_filter).fetchall()

    return jsonify(results)

app.run()
```

Save the code as `api_final.py` in your `api` folder and run it by navigating to your project folder in the terminal and entering the command:

```
python api_final.py
```

Once this example is running, try out the filtering functionality with these HTTP requests:

http://127.0.0.1:5000/api/v1/resources/books/all
http://127.0.0.1:5000/api/v1/resources/books?author=Connie+Willis
http://127.0.0.1:5000/api/v1/resources/books?author=Connie+Willis&published=1999
http://127.0.0.1:5000/api/v1/resources/books?published=2010

The database downloaded for this lesson has 67 entries, one for each of the winners of the Hugo Award for best science fiction novel between 1953 and 2014 (avoiding the voting controversy of 2015). The data set includes the novel's title, author, year of publication, and first sentence. Our API allows users to filter by three fields: `id`, `published` (year of publication), and `author`.

The first request returns all entries in the database, similar to the `/all` request we implemented for the last version of our API. The second request returns all books by the author Connie Willis (`?author=Connie+Willis`). Note that, within a query parameter, spaces between words are denoted with a `+` sign, hence `Connie+Willis`. The third request filters by two fields—author and year of publication. Instead of the three books returned by requesting `?author=Connie+Willis`, this request returns only the entry to *The Doomsday Book*, published in 1993. The last request returns all Hugo winners from the year 2010 (note that, in some years, more than one Hugo is awarded).

As we can see this version of our API serves a larger number of results, results that are stored in an SQLite database (`books.db`). When our user requests an entry or set of entries, our API pulls that information from the database by building and executing an SQL query. This iteration of our API also allows for filtering by more than one field. We'll discuss potential uses for this functionality after examining our code more closely.

Understanding Our Database-Powered API

Relational databases allow for the storage and retrieval of data, which is stored in tables. Tables are similar to spread sheets in that they have columns and rows—columns indicate what the data representes, such as "title" or "date." Rows represent individual entries, which could be books, users, transactions, or any other kind of entity.

The database we're working with has five columns `id`, `published`, `author`, `title`, and `first_sentence`. Each row represents one book that won the Hugo award in the year under the `published` heading, and the text of which begins with the sentence in the `first_sentence` column.

Rather than use test data defined in the application, our `api_all` function pulls in data from our Hugo database:

```
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM books;').fetchall()

    return jsonify(all_books)
```

First, we connect to the database using our `sqlite3` library. An object representing the connection to the database is bound to the `conn` variable. The `conn.row_factory = dict_factory` line lets the connection object know to use the `dict_factory` function we've defined, which returns items from the database as dictionaries rather than lists—these work better when we output them to JSON. We then create a cursor object (`cur = conn.cursor()`), which is the object that actually moves through the database to pull our data. Finally, we execute an SQL query with the `cur.execute` method to pull out all available data (`*`) from the `books` table of our database. At the end of our function, this data is returned as JSON: `jsonify(all_books)`. Note that our other function that returns data, `api_filter`, will use a similar approach to pull data from the database.

The purpose of our `page_not_found` function is to create an error page seen by the user if the user encounters an error or inputs a route that hasn't been defined:

```
@app.errorhandler(404)
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not be found.</p>", 404
```

In HTML responses, the code `200` means "OK"(the expected data transferred), while the code `404` means "Not Found" (there was no resource available at the URL given). This function allows us to return 404 pages when something goes wrong in the application.

Our `api_filter` function is an improvement on our previous `api_id` function that returns a book based on its ID. This new function allows for filtering by three different fields: `id`, `published`, and `author`. The function first grabs all the query parameters provided in the URL (remember, query parameters are the part of the URL that follows the `?`, like `?id=10`).

```
query_parameters = request.args
```

It then pulls the supported parameters `id`, `published`, and `author` and binds them to appropriate variables:

```
id = query_parameters.get('id')
published = query_parameters.get('published')
author = query_parameters.get('author')
```

The next segment begins to build an SQL query that will be used to find the requested information in the database. SQL queries used to find data in a database take this form:

```
`SELECT <columns> FROM <table> WHERE <column=match> AND <column=match>;
```

To get the correct data, we need to build both an SQL query that looks like the above and a list with the filters that will be matched. Combined, the query and the the filters provided by the user will allow us to pull the correct books from our database.

We begin to define both the query and the filter list:

```
query = "SELECT * FROM books WHERE"
to_filter = []
```

Then, if `id`, `published`, or `author` were provided as query parameters, we add them to both the query and the filter list:

```
    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
    if author:
        query += ' author=? AND'
        to_filter.append(author)
```

If the user has provided none of these query parameters, we have nothing to show, so we send them to the "404 Not Found" page:

```
    if not (id or published or author):
        return page_not_found(404)
```

To perfect our query, we remove the trailing `AND` and cap the query with the `;` required at the end of all SQL statements:

```
    query = query[:-4] + ';'
```

Finally, we connect to our database as in our `api_all` function, then execute the query we've built using our filter list:

```
conn = sqlite3.connect('books.db')
conn.row_factory = dict_factory
cur = conn.cursor()

results = cur.execute(query, to_filter).fetchall()
```

Finally, we return the results of our executed SQL query as JSON to the user:

```
return jsonify(results)
```

Whew! When all is said and done, this section of code reads query parameters provided by the user, builds an SQL query based on those parameters, executes that query to find matching books in the database, and returns those matches as JSON to the user. This section of code makes our API's filtering capability considerably more sophisticated—users can now find books by, for example, Ursula K. Le Guin that were published in 1975 or all books in the database published in 2010.

Our API in Practice

Now that we have implemented our Distant Reading API, let's consider how it might be of use in digital projects and in research.

One of the advantages of providing data through an API, as opposed to providing some kind of downloadable database or file for users, is that, as new data or additional resources are added, they become immediately available to projects built using the API. Imagine that we make our API publicly available, and a user creates a visualization that plots the length of a novel's first sentence in characters against its year of publication:

{% include figure.html filename="hugo.png" caption="Scatterplot of first sentence length against date of publication." %}

As new Hugo winners were added to the database, the script that generated this visualization would immediately be able to use the new information. If the visualization were created in D3 or another web-based utility, this plot would actually reflect additional data added to the book archive as soon as the archive was updated—that is, in real time. As additional data accrued, we might, for example, learn if John Scalzi's unusually lengthy opening to his 2013 *Red Shirts* was an aberration or the continuation of a longer trend toward wordiness in science fiction.

A strong API can be considered the backbone of a potentially limitless number of projects or avenues of research. Though the above example takes the form

of a visualization of the limited amount of data we've provided in our Distant Reading Archive, a project based on this API might just as easily take the form of a Twitterbot that shares first sentences or a library webpage that displays book openings and year of publication alongside other book metadata. In many cases, it makes sense to first create an API interface to your core data or functionality before extrapolating on it to create a visualization, application, or website. Not only does it make your work accessible to researchers working on other projects, but it often leads to a more comprehensible and maintainable project.

Resources

The below resources provide information on useful APIs for researchers in the humanities and social sciences as well as further reading on API concepts.

APIs for Humanities Researchers

Chronicling America (Library Of Congress)

A digitized collection of American newspaper articles from the 18th to the 20th century.

Connecting Repositories (CORE)

A collection of open access articles from various sources hosted by the Open University.

English Broadside Ballad Archive (EBBA) https://diggingintodata.org/repositories/english-broadside-ballad-archive-ebba

History Data Service (HDS)

A collection of data from a wide variety of historical sources.

ARTstor

More than two million images from historical, carographic, and art history sources.

Digging into Data API List https://diggingintodata.org/repositories

API Resources

Anatomy of a URL

Explains the different sections of a URL (protocol, domain, path, and so on) in greater detail.

Original Paper on REST

PhD thesis by Roy Thomas Fielding that introduced the concepts behind the REST philosophy of API design.

The Flask Mega Tutorial

The most well-known tutorial for learning the Flask web framework.

Reading 4

Chronicling America API Reference

About the Site and API

Introduction

Chronicling America provides access to information about historic newspapers and select digitized newspaper pages. To encourage a wide range of potential uses, we designed several different views of the data we provide, all of which are publicly visible. Each uses common Web protocols, and access is not restricted in any way. You do not need to apply for a special key to use them. Together they make up an extensive application programming interface (API) which you can use to explore all of our data in many ways.

Details about these interfaces are below. In case you want to dive right in, though, we use HTML link conventions to advertise the availability of these views. If you are a software developer or researcher or anyone else who might be interested in programmatic access to the data in Chronicling America, we encourage you to look around the site, "view source" often, and follow where the different links take you to get started. When describing Chronicling America as the source of content, please use the URL and a Web site citation, such as "from the Library of Congress, Chronicling America: Historic American Newspapers site".

For more information about the open source Chronicling America software please see the LibraryOfCongress/chronam GitHub site. Also, please consider subscribing to the ChronAm-Users discussion list if you want to discuss how to use or extend the software or data from its APIs.

If you're interested in other data and machine-usable interfaces available from the Library of Congress, you might find the LC for Robots page helpful at https://labs-dev.loc.gov/lc-for-robots/++

The API

Jump to:

- Search the newspaper directory and digitized page contents using OpenSearch.
- Auto Suggest API for looking up newspaper titles
- Link using our stable URL pattern for Chronicling America resources.
- JSON views of Chronicling America resources.
- Linked Data views of Chronicling American resources.
- Bulk Data for research and external services.
- CORS and JSONP support for your JavaScript applications.

Searching the directory and newspaper pages using OpenSearch

The directory of newspaper titles contains nearly 140,000 records of newspapers and libraries that hold copies of these newspapers. The title records are based on MARC data gathered and enhanced as part of the NDNP program.

Searching the title records is possible using the OpenSearch protocol . This is advertised in a LINK header element of the site's HTML template as "NDNP Title Search", using this OpenSearch Description document.

Title search parameters:

- terms: the search query
- format: 'html' (default), 'json', or 'atom' (optional)
- page: for paging results (optional)

Examples:

Note that all example URLs below use the same protocol and server name, http://chroniclingamerica.loc.gov/. We only show the URL paths and parameters below to save space.

- /search/titles/results/?terms=michigan
  search for "michigan", HTML response
- /search/titles/results/?terms=michigan&format=atom
  search for "michigan", Atom response
- /search/titles/results/?terms=michigan&format=json&page=5
  search for "michigan", JSON response, starting at page 5

There are millions of digitized newspaper pages in Chronicling America. These pages span several decades and many U.S. states and territories. New batches of data come in from partner institutions throughout the year and are added to the site regularly.

Searching newspaper pages is also possible via OpenSearch. This is advertised in a LINK header element of the site's HTML template as "NDNP Page Search", using this OpenSearch Description document.

Page search parameters:

- andtext: the search query
- format: 'html' (default), or 'json', or 'atom' (optional)
- page: for paging results (optional)

Examples:

- /search/pages/results/?andtext=thomas
  search for "thomas", HTML response
- /search/pages/results/?andtext=thomas&format=atom
  search for "thomas", Atom response
- /search/pages/results/?andtext=thomas&format=atom&page=11
  search for "thomas", Atom response, starting at page 11

Using the Directory's AutoSuggest API

The Chronicling America Directory contains hundreds of thousands of bibliographic records for American newspaper titles. To allow the directory to be integrated into your own applications you can use the OpenSearch AutoSuggest API to dynamically lookup these newspaper titles. For example:

- /suggest/titles/?q=Florida

The response will be application/x-suggestions+json as described by the the OpenSearch Suggestions extension.

Link to Chronicling America Resources

The Chronicling America Web site uses links that follow a straightforward pattern. You can use this pattern to construct links into specific newspaper titles, to any of its available issues and their editions, and even to specific pages. These links can be readily bookmarked and shared on other sites.

We are committed to supporting this link pattern over time, so even if we change how the site works, we will redirect any requests to the system using this specific pattern into the new site. We established redirect rules for links into the previous version of the site when we released a new version in early 2009, and we intend to sustain those rules.

The link pattern uses LCCNs, dates, issue numbers, edition numbers, and page sequence numbers.

Examples:

- /lccn/sn86069873/
  title information for LCCN sn 86069873
- /lccn/sn86069873/issues/
  calendar view of available issues
- /lccn/sn86069873/issues/first_pages/
  browse first pages of available issues
- /lccn/sn86069873/1900-01-05/ed-1/
  first available edition from January 5, 1900
- /lccn/sn86069873/1900-01-05/ed-1/seq-3/
  third available page from first edition, January 5, 1900

JSON Views

In addition to the use of JSON in OpenSearch results, there are also JSON views available for various resources in Chronicling America. These JSON views are typically linked from their HTML representation using the <link> element. For example:

- /lccn/sn86069873.json
  title information for LCCN sn 86069873
- /lccn/sn86069873/1900-01-05/ed-1.json
  first available edition from January 5, 1900
- /lccn/sn86069873/1900-01-05/ed-1/seq-3.json
  third available page from first edition, January 5, 1900
- /newspapers.json
  a list of all newspaper titles for which there is digital content
- /batches.json
  a list of all batches of content that have been loaded
- /batches/batch_dlc_jamaica_ver01.json
  detailed information about a specific batch
- /awardees.json
  a list of all NDNP Awardees as JSON
- /awardees/dlc.json
  detailed information about a specific awardee

You will notice that many of these JSON views link to each other. This allows the JSON to stay relatively compact by including only information about a particular entity, while providing a link to fetch more detail about a related entity.

Linked Data

Linked Data allows us to connect the information in Chronicling America directly to related data on the Web explicitly. Chronicling America provides several Linked Data views to make it easy to connect with other information resources and to process and analyze newspaper information with conceptual precision.

We use concepts like Title (defined in DCMI Metadata Terms) and Issue (defined in the Bibliographic Ontology) to describe newspaper titles and issues available in the data. Using these concepts, defined in existing ontologies, can help to ensure that what we mean by "title" and "issue" is consistent with the intent of other publishers of linked data. We also define other concepts not already defined in existing ontologies. This vocabulary includes elements suitable for newspaper information and the NDNP program, including these elements:

- Awardee
- Batch

- Page
- bag
- number
- section
- sequence

These elements are used in RDF views of several types of pages, ranging from a list of the newspaper titles available on the site and information about each, to enumerations of all the pages that make up each issue and all of the files available for each page.

Examples:

- /lccn/sn85038615.rdf
  information about The Times Dispatch (Richmond, Va.) 1903-1914
- /lccn/sn83045555/1889-11-21/ed-1.rdf
  information about the November 21, 1889 first edition issue of Deseret Evening News (Great Salt Lake City [Utah])
- /lccn/sn83030214/1905-01-15/ed-1/seq-25.rdf
  details about all of the files associated with the 25th image of the January 15, 1905 issue of New-York Tribune (New York [N.Y.]) 1886-1924
- /awardees/mohi.rdf
  information about awardee State Historical Society of Missouri
- /newspapers.rdf
  list of available newspaper titles

Comparing the RDF versions of the links above with their HTML counterpart links, you might notice that the URI pattern we follow for these views is to remove the final slash, replacing it with ".rdf". We follow this pattern to comply with best practices for publishing linked data, and also to keep the URIs easy to understand and use.

For each of the HTML pages with a linked data counterpart in RDF, we provide links to those alternate views from the HTML page using the LINK header element. This can support automating the process of using the RDF data in tools like bookmarklets, plugins, and scripts, and it also helps us to advertise the availability of the additional views. In many views, such as newspaper page images, we also provide LINK elements pointing to the various available files (image, text, OCR coordinate XML) for each available page or other potentially useful information. We encourage you to explore the entire site and to look for and use these LINK elements if you find them useful when working with NDNP data. Just follow your nose, and view the source.

In addition to the concepts describe above, we use concepts from several other vocabularies in describing NDNP materials and also in linking to related data available on other sites. These additional vocabularies and external sites include:

- DBpedia
- Dublin Core and DCMI Terms
- FRBR concepts in RDF
- GeoNames
- LCCN Permalink
- lingvoj.org
- OAI-ORE (more about aggregations below)
- OWL
- RDA
- WorldCat

We are grateful to all of these providers and we hope we can follow their lead in encouraging additional connections between data and vocabulary providers. Please be aware that how we use these vocabularies will likely change over time, as they continue to develop, and as new vocabularies are introduced.

Bulk Data

In certain situations the granular access provided by the API may be somewhat constraining. For example, perhaps you are a researcher who would like to try out new indexing techniques on the millions of pages of OCR data in Chronicling America. Or perhaps you are a service provider and anticipate needing to support a high volume of fulltext searches across the corpus, and do not want the Chronicling America API as an external dependency. To support these and other potential use cases we are beginning to provide bulk access to the underlying data sets. The initial bulk data sets include:

- Batches: each batch of digitized content that is provided by awardees is made available via the Batches HTML, Atom and JSON views. These views provide links to where the files comprising the batch can be fetched with a web crawling tool like wget.
- OCR Bulk Data: the complete set of OCR XML and text files that make up the newspaper collection are made available as compressed archive files. These files are listed in the OCR report, and are also made available via Atom and JSON feeds that will allow you to build automated workflows for updating your local collection.

CORS and JSONP Support

To help you integrate Chronicling America into your JavaScript applications, the OpenSearch and AutoSuggest JSON responses support both Cross-Origin Resource Sharing (CORS) and JSON with Padding (JSONP). CORS and JSONP allow your JavaScript applications to talk to services hosted at chroniclingamerica.loc.gov without the need to proxy the requests yourself.

CORS Example

```
curl -i 'http://chroniclingamerica.loc.gov/suggest/titles/?q=manh'

HTTP/1.1 200 OK
Date: Mon, 28 Mar 2011 19:45:34 GMT
Expires: Tue, 29 Mar 2011 19:45:37 GMT
ETag: "7d786bec2ca003d86009f8ccdfd72912"
Cache-Control: max-age=86400
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: X-Requested-With
Content-Length: 7045
Last-Modified: Mon, 28 Mar 2011 19:45:37 GMT
Content-Type: application/x-suggestions+json
```

```
[
  "manh",
    [
      "Manhasset life. (Manhasset, N.Y.) 19??-19??",
      "Manhasset mail. (Manhasset, N.Y.) 1927-1986"
    ],
    [
      "sn97063690",
      "sn95071148"
    ],
    [
      "http://chroniclingamerica.loc.gov/lccn/sn97063690/",
      "http://chroniclingamerica.loc.gov/lccn/sn95071148/"
    ]
]
```

JSONP Example

```
curl -i 'http://chroniclingamerica.loc.gov/suggest/titles/?q=manh&callback=suggest'

HTTP/1.1 200 OK
Date: Mon, 28 Mar 2011 19:45:34 GMT
Expires: Tue, 29 Mar 2011 19:45:37 GMT
ETag: "7d786bec2ca003d86009f8ccdfd72912"
Cache-Control: max-age=86400
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: X-Requested-With
Content-Length: 7045
Last-Modified: Mon, 28 Mar 2011 19:45:37 GMT
Content-Type: application/x-suggestions+json

suggest([
  "manh",
    [
      "Manhasset life. (Manhasset, N.Y.) 19??-19??",
      "Manhasset mail. (Manhasset, N.Y.) 1927-1986"
    ],
    [
      "sn97063690",
      "sn95071148"
    ],
    [
      "http://chroniclingamerica.loc.gov/lccn/sn97063690/",
      "http://chroniclingamerica.loc.gov/lccn/sn95071148/"
    ]
]);
```

CORS is arguably a more elegant solution, and is supported by most modern browsers. However JSONP might be a better option if your application needs legacy browser support.

Reading 5

API University Introduction

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

APIs Are Like User Interfaces--Just With Different Users in Mind

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

Whenever the mainstream business media starts to cover a technology as though it is some sort of wonder drug (like it has for APIs, or application programming interfaces), it creates a groundswell of curiosity. People want to know what exactly the wonder drug is? How does it work? How might it benefit them? Where can they get some?

This series is designed to answer some of these basic questions for all readers, regardless of their level of technical expertise or perspective (business, technical, consumer, and so on). It is not meant to be the end-all, be-all encyclopedic explanation of APIs. Rather, it is meant to give readers the basics—to make you smart enough to intelligently participate in a conversation about APIs. We envision this series as a living document that we will add to and improve over time. We welcome your feedback with that objective in mind.

Perhaps the best place to start in this introduction to our series (/api-university/what-are-apis-and-how-do-they-work) is to simply explain APIs as an alternative to something that anybody who has used a desktop or mobile application is already familiar with: the user interface.

For decades, most computer software -- programs that provide data like contact information or functionality like image editing -- has been conceived and offered with one type of user in mind: a human. No matter what chain of events was taking place under the hood of that software, a human user was traditionally at the end of that chain. That end user invariably consumed that data and/or functionality through a user interface (UI)—an experience that's designed to make that act of consumption as easy and even enjoyable as possible.

But what if that same data and/or functionality could be just as easily consumed by another piece of software? In this case, the UI concerns are very different. After all: Software doesn't have eyes, emotions or intuition, so it doesn't need an enjoyable, intuitive graphical user interface. However, much the same way a UI is tailored to humans, software needs an alternative interface that makes it easy to consume data and/or functionality.

Enter, application programming interfaces, or APIs.

An API is very much the same thing as a UI, except that it is geared for consumption by software instead of humans. This is why APIs are often explained in the mainstream media as a technology that allows applications (software programs) to talk to one another. In this context, the terms "software," "applications," "machines" and "computers" are virtually interchangeable. For example, APIs are often discussed as being machine-readable interfaces (versus human-readable).

The significance of APIs to the modern world should not be underestimated. With each day, their importance to three primary constituencies—customers (the public), businesses (and business-like organizations such as government and non-profits) and programmers--seems to grow. Starting about 2005, (the same year that *ProgrammableWeb* was founded), the groundswell of interest in APIs has given birth to a cottage industry that isn't so cottage anymore: the API economy.

Many people will be satisfied with the information we have provided in this report so far. But if you want to know more about how APIs work, what their benefits are, and how they compare to something we use in the physical world every day, be sure to read the remaining parts of this series starting with "What Exactly Is an API? (/news/what-exactly-api/analysis/2015/12/03)"

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

What is an API, Exactly?

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

This is the first part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work). If you missed the introduction to this series -- APIs Are Like User Interfaces--Just With Different Users In Mind (/news/apis-are-user-interfaces-just-different-users-mind/analysis/2015/12/03) -- we strongly suggest you start there first. In this second part of our series, we will define the million dollar question "What is an API?"

Another way of thinking about APIs is in the context of a wall socket. In North America and other parts of the world, the electrical sockets found in the walls of homes and businesses are essentially interfaces to a service: the electricity that's consumed by everything from our computers and smartphones to appliances like vacuum cleaners and hair dryers. In other words, it's important to think of the electricity as a service and of each device that uses electricity as a consumer of that service. In this scenario, instead of having its own source of power (the way a car does), the consumer is outsourcing its power requirements to the provider of a service.

While they may differ depending on where they are in the world, electrical sockets have predictable patterns of openings into which electrical plugs that match those patterns fit. The service itself also conforms to certain specifications. For example, in North America, most wall sockets deliver 120 volts of alternating current (AC) operating at 60Hz (the frequency at which the current alternates directions). This specification essentially sets an expectation on behalf of consuming devices. For example, your microwave oven has an expectation of the wall socket that it's plugged-into and the wall-socket has an expectation of the microwave oven. Likewise, an API specifies how software components should interact with each other.

So, what, if anything do electric wall sockets have to do with APIs? Well, among the many metaphors that could be used to help visualize how APIs work, the wall socket comes pretty darn close and in the next part of the series, we'll explain why.

This is a part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In the next part (part 2 (/news/what-are-benefits-apis/analysis/2015/12/03)), we'll use the metaphor of the electric wall socket to explain some of the key benefits of APIs.

About the author: David Berlind (/profile/davidberlind) is the editor-in-chief of ProgrammableWeb.com (https://www.programmableweb.com). You can reach him at david.berlind@programmableweb.com (mailto:david.berlind@programmableweb.com). Connect to David on Twitter at @dberlind (http://twitter.com/dberlind) or on LinkedIn (https://www.linkedin.com/in/davidberlind), put him in a Google+ (https://plus.google.com/+DavidBerlind?rel=author) circle, or friend him on Facebook (https://www.facebook.com/david.berlind).

@dberlind (http://twitter.com/dberlind) (https://www.facebook.com/david.berlind) (https://plus.google.com/+DavidBerlind) (http://www.linkedin.com/in/davidberlind/) (mailto:david.berlind@programmableweb.com)

‹ Google Cloud Platform Updates Include Custom Machine Types (/news/google-cloud-platform-updates-include-custom-machine-types/2015/12/01)

What are the Benefits of APIs? ›

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

APIs Are Like User Interfaces--Just With Different Users in Mind

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

Whenever the mainstream business media starts to cover a technology as though it is some sort of wonder drug (like it has for APIs, or application programming interfaces), it creates a groundswell of curiosity. People want to know what exactly the wonder drug is? How does it work? How might it benefit them? Where can they get some?

This series is designed to answer some of these basic questions for all readers, regardless of their level of technical expertise or perspective (business, technical, consumer, and so on). It is not meant to be the end-all, be-all encyclopedic explanation of APIs. Rather, it is meant to give readers the basics—to make you smart enough to intelligently participate in a conversation about APIs. We envision this series as a living document that we will add to and improve over time. We welcome your feedback with that objective in mind.

Perhaps the best place to start in this introduction to our series (/api-university/what-are-apis-and-how-do-they-work) is to simply explain APIs as an alternative to something that anybody who has used a desktop or mobile application is already familiar with: the user interface.

For decades, most computer software -- programs that provide data like contact information or functionality like image editing -- has been conceived and offered with one type of user in mind: a human. No matter what chain of events was taking place under the hood of that software, a human user was traditionally at the end of that chain. That end user invariably consumed that data and/or functionality through a user interface (UI)—an experience that's designed to make that act of consumption as easy and even enjoyable as possible.

But what if that same data and/or functionality could be just as easily consumed by another piece of software? In this case, the UI concerns are very different. After all: Software doesn't have eyes, emotions or intuition, so it doesn't need an enjoyable, intuitive graphical user interface. However, much the same way a UI is tailored to humans, software needs an alternative interface that makes it easy to consume data and/or functionality.

Enter, application programming interfaces, or APIs.

An API is very much the same thing as a UI, except that it is geared for consumption by software instead of humans. This is why APIs are often explained in the mainstream media as a technology that allows applications (software programs) to talk to one another. In this context, the terms "software," "applications," "machines" and "computers" are virtually interchangeable. For example, APIs are often discussed as being machine-readable interfaces (versus human-readable).

The significance of APIs to the modern world should not be underestimated. With each day, their importance to three primary constituencies—customers (the public), businesses (and business-like organizations such as government and non-profits) and programmers--seems to grow. Starting about 2005, (the same year that *ProgrammableWeb* was founded), the groundswell of interest in APIs has given birth to a cottage industry that isn't so cottage anymore: the API economy.

Many people will be satisfied with the information we have provided in this report so far. But if you want to know more about how APIs work, what their benefits are, and how they compare to something we use in the physical world every day, be sure to read the remaining parts of this series starting with "What Exactly Is an API? (/news/what-exactly-api/analysis/2015/12/03)"

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

What is an API, Exactly?

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

This is the first part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work). If you missed the introduction to this series -- APIs Are Like User Interfaces--Just With Different Users In Mind (/news/apis-are-user-interfaces-just-different-users-mind/analysis/2015/12/03) -- we strongly suggest you start there first. In this second part of our series, we will define the million dollar question "What is an API?"

Another way of thinking about APIs is in the context of a wall socket. In North America and other parts of the world, the electrical sockets found in the walls of homes and businesses are essentially interfaces to a service: the electricity that's consumed by everything from our computers and smartphones to appliances like vacuum cleaners and hair dryers. In other words, it's important to think of the electricity as a service and of each device that uses electricity as a consumer of that service. In this scenario, instead of having its own source of power (the way a car does), the consumer is outsourcing its power requirements to the provider of a service.

While they may differ depending on where they are in the world, electrical sockets have predictable patterns of openings into which electrical plugs that match those patterns fit. The service itself also conforms to certain specifications. For example, in North America, most wall sockets deliver 120 volts of alternating current (AC) operating at 60Hz (the frequency at which the current alternates directions). This specification essentially sets an expectation on behalf of consuming devices. For example, your microwave oven has an expectation of the wall socket that it's plugged-into and the wall-socket has an expectation of the microwave oven. Likewise, an API specifies how software components should interact with each other.

So, what, if anything do electric wall sockets have to do with APIs? Well, among the many metaphors that could be used to help visualize how APIs work, the wall socket comes pretty darn close and in the next part of the series, we'll explain why.

This is a part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In the next part (part 2 (/news/what-are-benefits-apis/analysis/2015/12/03)), we'll use the metaphor of the electric wall socket to explain some of the key benefits of APIs.

About the author: David Berlind (/profile/davidberlind) is the editor-in-chief of ProgrammableWeb.com (https://www.programmableweb.com). You can reach him at david.berlind@programmableweb.com (mailto:david.berlind@programmableweb.com). Connect to David on Twitter at @dberlind (http://twitter.com/dberlind) or on LinkedIn (https://www.linkedin.com/in/davidberlind), put him in a Google+ (https://plus.google.com/+DavidBerlind?rel=author) circle, or friend him on Facebook (https://www.facebook.com/david.berlind).

@dberlind (http://twitter.com/dberlind) (https://www.facebook.com/david.berlind) (https://plus.google.com/+DavidBerlind) (http://www.linkedin.com/in/davidberlind/) (mailto:david.berlind@programmableweb.com)

‹ Google Cloud Platform Updates Include Custom Machine Types (/news/google-cloud-platform-updates-include-custom-machine-types/2015/12/01)

What are the Benefits of APIs? ›

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

What are the Benefits of APIs?

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

This is the second part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In part (/news/what-exactly-api/analysis/2015/12/03) 1 (/news/what-exactly-api/analysis/2015/12/03), we used the standard electrical socket found in most walls as a metaphor for explaining the principles of an API.

Imagine what life might be like without such a standard. For example, with no plug, matching socket or standard particulars (often called specifications), we might have to hard-wire appliances into the walls of buildings. This would involve gathering the required tools, unsheathing all the wires and splicing the wires together. Of course, we would also need to know something about the wires coming out of the wall. Moving an appliance from one location to another would be a major undertaking in disconnection and reconnection.

Luckily, we don't have to do any of this. We have plugs and sockets that conform to a standard, which affords the following "conveniences":

- Through the standard interface, any compatible consumer (in this case, a device) can easily outsource its electrical requirements to the service, and the device can expect to get the same results. Knowing that electricity can be outsourced through a predictably available standard interface, device manufacturers can focus on making great devices and not on figuring out how those devices are going to get their power (other than supporting the standard interface).
- Consuming-devices are easily moved from one socket to another. Thanks to a standard interface, moving a hair dryer from a home in Boston to a hotel in San Francisco is no different than moving it from your bedroom to your kitchen. And even if the pattern of the standard interface changes —as it does when traveling from North America to the United Kingdom--consuming devices can be easily adapted.
- The electrical socket interface is a layer of abstraction (it hides the specifics) to the underlying service. The consumer (again, not a person but rather a device) is blind to things like the color of the wiring in the walls, other devices the wiring is shared with, how the electricity is generated (whether it be from a wind farm, nuclear plant, coal-fired generator or solar array) or where those sources of power are located. For whatever motivations (such as cost savings, policy or public pressure), so long as the service delivers 120 volts of 60Hz AC power to the wall socket in a way that conforms to the standard, the service provider is free to change anything and everything from just behind the socket all the way to the source of power. Any such changes are transparent to consuming devices.
- The same transparency works both ways. To the service, all consumers look the same. It is essentially blind to the specifics of the consuming device that's on the other side of the electrical socket.

As interfaces go, APIs and the benefits they provide to their consumers (such as desktop, Web, mobile and server-side applications) are not that much different from electrical sockets and the benefits they provide to their consumers (such as computers and appliances).

For example, in much the same way a consuming device can outsource its electricity to a service through a wall socket interface, a consuming application can outsource its requirements for data (such as a patient record) or functionality (such as a location represented as a pin on an interactive map) to a service through an application programming interface. Though the consuming application and service (known as an API provider) are sufficiently decoupled to the point that they know little or nothing about one another, the interface through which they communicate represents a set of agreed-upon standards (similar to the 120v/60Hz AC standard for electricity) that enables applications to make requests of the service and get data and/or functionality in return.

And, similar to the way in which consuming devices outsource their power requirements to service providers through an electrical network, applications can outsource their requirements for data and functionality to service providers that are across digital networks, like the Internet (or even just across a local-area network at a business). For example, a mobile application for home buyers can incorporate interactive maps and navigation into its user experience by outsourcing that functionality to Google Maps. Each time that mobile application displays a new interactive map, it does so by sending a request across the Internet to a special API that's offered by Google for that very purpose.

So, as with electricity, APIs can be consumed (to the extent that an application outsources certain requirements to an API) and provided as a service.

In addition to calling APIs from across a network, application developers can leverage APIs offered by the local system or device that their application runs on. For example, applications can discover a smartphone's current location by calling the API associated with the phone's GPS receiver. There's also an emerging class of APIs offered by modern Web browsers, such as Chrome and Firefox, that are like the aforementioned standard electrical sockets--across all types of systems (desktops, tablets, smartphones, and so on), these browsers offer a standard way for browser-based applications to access the host device's storage, audio system, cameras, and much more. From across a network, the source of data or functionality that's called through an API could be an application server (a database server, for example), a translation service or even a network-enabled refrigerator.

This is part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In part 3 (/news/how-web-and-browser-apis-fuel-api-economy/analysis/2015/12/03), we'll discuss the types of APIs that make the Web programmable (as in the "ProgrammableWeb").

About the author: David Berlind (/profile/davidberlind) is the editor-in-chief of ProgrammableWeb.com (https://www.programmableweb.com). You can reach him at david.berlind@programmableweb.com (mailto:david.berlind@programmableweb.com). Connect to David on Twitter at @dberlind (http://twitter.com/dberlind) or on LinkedIn (https://www.linkedin.com/in/davidberlind), put him in a Google+ (https://plus.google.com/+DavidBerlind?rel=author) circle, or friend him on Facebook (https://www.facebook.com/david.berlind).

@dberlind (http://twitter.com/dberlind) (https://www.facebook.com/david.berlind) (https://plus.google.com/+DavidBerlind) (http://www.linkedin.com/in/davidberlind/) (mailto:david.berlind@programmableweb.com)

‹ What is an API, Exactly? (/news/what-api-exactly/analysis/2015/12/03)

How Web and Browser APIs Fuel The API Economy ›

Find out more about... API Education (/category/api-education)

COMMENTS (0)

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

How Web and Browser APIs Fuel The API Economy

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

This is the third part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In part (/news/what-are-benefits-apis/analysis/2015/12/03) 2 (/news/what-are-benefits-apis/analysis/2015/12/03), we covered some of the key benefits of APIs.

First, let's review the primary constituencies impacted by APIs, starting with computer programmers, also known as application developers.

APIs Make Developers More Productive

When developers write code, rarely do they start from scratch. For example, if a software company is going to create a word processor that runs on Microsoft Windows, the developers of that word processor would make use of various features already built into Windows instead of attempting to recreate those features.

This begins with the most basic part of Windows--the windows themselves. Prior to the existence of Windows and other graphical operating systems, if a programmer wanted to present a resizable box on a screen complete with a title bar and buttons for minimizing, maximizing or closing that box, s/he would have to write thousands of lines of code from scratch. From one application to the next, this "windowing" capability might work differently, as each programmer might have a different idea of how such windows should look and feel. But when Microsoft began to offer Windows as an operating system, it also provided developers with a means--an API--for drawing those windows on the screen with only a few minutes of work.

In this context, Microsoft provided the API as means to access the windowing service in the Windows operating system, and the developers of the thousands of applications that run on Windows "consumed" that service through its API. Those developers didn't have to write code to draw the title bar of the window nor provide for the resizing features of the window. Instead, those features were inherited by any window that was created using the windowing API found in Microsoft Windows itself.

Likewise, programmers don't have to write thousands of lines of code for storing individual bits and bytes onto a computer's hard drive—Microsoft Windows includes a specialized API for doing that, as well. Within their code, programmers simply make a reference to that API (more commonly referred to as "calling an API"), supplying it with the data they want Windows to save to the hard drive.

Many APIs expect such inputs, otherwise known as parameters. The windowing API, for example, requires certain parameters; it needs to be fed the exact coordinates of the top left and bottom right corners of the window its about to paint. Then, you guessed it, the windowing service in Windows takes care of the rest.

In much the same way that the electrical socket abstracts the complexities of the electrical service that, say, your vacuum cleaner consumes, when APIs hide all of the complexities, internals and logic required to complete a task (such as painting a window, saving data to a hard drive, translating a word, reading the current location from a GPS receiver or presenting an interactive map), those APIs are serving as a layer of abstraction to the underlying service.

The extent to which APIs make oft-repeated yet complex processes highly re-usable with just a single or few lines of code is fundamental to developer productivity, modern day application development and the API economy.

Using this kind of a model, programmers are significantly more productive than they would be if they had to write the code from scratch. They don't have to "reinvent the wheel" with every new program they write. They can instead focus on the unique proposition of their applications while outsourcing all of the commodity functionality to APIs.

Take credit card processing, for example. There isn't a whole lot of room for innovation in this area-- it is unlikely that custom-written credit card processing routines are going to make, say, one taxi cab service more competitive than the next. It would therefore be a waste of time for developers working for the taxi cab service to build a credit card processing service from scratch. The return on investment compared to simply outsourcing that credit card processing to a best-of-breed API provider (like Stripe) makes the custom-code alternative financially infeasible.

The API economy's growth is thusly driven by service providers that compete to address this thirst for greater developer productivity by packaging commodity and, often, complex functionality into easily reused API-based components. For each of the various types of functionality that can be invoked via API (such as credit card processing, mapping, navigation and translation), there are often multiple API providers vying for the attention of application developers. In turn, as more componentry is supplied in the form of API-based services, the API economy is accelerating the trend toward a world of applications that are primarily composed from off-the-shelf APIs.

The net result of the API economy's vicious and accelerating circle is that applications that once took months or years to build now take days or weeks or even hours. Developers are not only more productive, but the time it takes a business to make an application available to its customers is dramatically reduced. In turn, those customers benefit from shorter development cycles because the applications they use are being updated with new and innovative features more frequently.

Networkable APIs are the game-changers

As has just been suggested, APIs aren't limited to Windows, nor are they confined to what can be found in the same system (such as a desktop, tablet or a server) that runs API-consuming applications. Much the same way an application can call a hard drive API from Windows, a taxi-dispatching application can call a credit card processing API like Stripe from across the Internet. In other words, developers can also consume APIs that are offered by remote systems and devices that are reachable through a network. That network can be a private network, like the ones found in homes and businesses, or, it can be a public network like the Internet.

Whereas the computer logic behind the API that saves the data to a hard drive is often local to the personal computer that's also running the API-consuming application, this type of networked API works a little bit differently. Using our taxi-dispatching application as an example, all the logic for processing credit cards runs on Stripe's remote systems which in turn are abstracted by Stripe's credit-card processing API.

Just as the electrical socket in the wall represents the endpoint of the electrical system into which electricity-consuming devices are plugged, APIs involve an endpoint—essentially, a socket into which consuming applications are plugged. The number and types of devices that can be plugged into electrical sockets are limited only by the imaginations of inventors (and the capacity of the utility), and, likewise, the number of applications that can take advantage of the functionality abstracted by an API's endpoint is limited only by the imaginations of developers and the capacity of the API provider's infrastructure. Case in point: It wasn't long after Google offered an API for Google Maps that thousands of third-party developers stepped forward with unique and innovative applications that consumed the API, incorporating Google's mapping functionality directly into their apps.

It is for this reason that APIs are often referred to as an engine of innovation. The Google Maps API is just one API that spawned a cycle of innovation that continues to this day. The founders of the Instagram photo-sharing service might never have succeeded had it not been for the Facebook API that allowed Instagram users to broadcast their latest photos to their friends on Facebook. It isn't hard to imagine the sort of innovation that will be inspired by the millions of APIs that will one day be a part of the API economy.

This is part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In part 4 (/news/how-api-principles-abstraction-benefit-api-providers/analysis/2015/12/03), we'll cover the API principles of abstraction and why they are so beneficial to API providers.

API U Series: What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work)

Why Did They Put the Web in Web APIs?

API UNIVERSITY (/API-UNIVERSITY) Analysis (/category/all/news?articletypes=analysis), API Education (/category/api-education)

Dec. 03 2015 By David Berlind (/profile/davidberlind) Editor in Chief, ProgrammableWeb

This is the fifth part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In part 4 (/news/how-api-principles-abstraction-benefit-api-providers/analysis/2015/12/03), we focused on the API concept of abstraction and the type of flexibility it creates for API providers.

While there are no rules or laws governing exactly how developers must connect their applications to an API from across a network or the Internet, several approaches have emerged as de facto standards for achieving such connections and transmitting and receiving payloads. For example, when it comes to applications connecting to APIs from across the Internet, the majority of API providers make such connections possible over the HTTP protocol (Hypertext Transfer Protocol)--otherwise known as the World Wide Web.

HTTP is the same protocol that makes it possible for Web browsers (as the consuming software) to send and retrieve information to and from Web sites (the services). For example, when you visit the Web site for the National Football League (NFL.com) with your Web browser, your Web browser is exchanging information with the NFL's Web server over HTTP. In fact, whether you are retrieving game scores or submitting a list of your favorite teams to NFL.com, the site is relying on a special set of HTTP commands called "verbs" (including get, put and post) that are purpose-built for such client/server exchanges of data.

Software relies on HTTP verbs in much the same way that Web browsers use HTTP verbs to exchange information with a Web site. For example, the NFL has mobile applications for iOS and Android that can retrieve game scores and maintain lists of favorite teams. But, instead of retrieving information in a way that's formatted for easy-to-read display in a Web browser, software like the NFL's mobile apps retrieve information in more of a machine-readable format that it can subsequently manipulate for display within the mobile app's user experience. In most cases, the service provider (the NFL, in ch software to connect to. That Web address is different from the one

API endpoints that are addressable over the Web and that support the HTTP command structure are often said to be "Web APIs." To the extent that Web or mobile applications can be primarily composed from calls to multiple Web-based APIs from multiple Web API providers (these composite applications are sometimes called "mashups"), the Web has turned into a programmable platform that's equally, if not more, powerful than programmable platforms including Windows, Mac and Linux (thus, the name of our site: ProgrammableWeb.com).

At this point, it's important to note that there are different types of Internet-based APIs. While some don't follow these exact conventions (some may not even rely on HTTP), the API fundamentals--whereby developers are able to find an API to handle some commodity task instead of taking weeks, months or even years to write their own code--are the same.

Also, whereas some APIs are designed to query or update a database, other APIs simply initiate a process. For example, Sony makes cameras that can be remotely controlled via a Web API through which developers can activate a camera's shutter. Still other APIs, as mentioned earlier, can add functionality into your application. For example, instead of writing millions of lines of code and licensing third-party mapping data in order to present an interactive map in an application, a developer can accomplish the same thing with about 10 lines of code that engage the API for Google Maps.

Depending on the volume of API calls, an API provider like Google might charge the application developer a fee for using the API (giving rise to the idea of an "API economy"). When that's the case, the application developer must weigh all the costs of using the API versus developing the functionality from scratch. Or, as is often the case in the API economy, the developer can seek out a more economical provider of a similar service.

This is part of our series What Are APIs and How Do They Work? (/api-university/what-are-apis-and-how-do-they-work) In the conclusion (/news/api-economy-delivers-limitless-possibilities/analysis/2015/12/03), we will look at the limitless possibilities delievered by the API economy.

About the author: David Berlind (/profile/davidberlind) is the editor-in-chief of ProgrammableWeb.com (https://www.programmableweb.com). You can reach him at david.berlind@programmableweb.com (mailto:david.berlind@programmableweb.com). Connect to David on Twitter at @dberlind (http://twitter.com/dberlind) or on LinkedIn (https://www.linkedin.com/in/davidberlind), put him in a Google+ (https://plus.google.com/+DavidBerlind?rel=author) circle, or friend him on Facebook (https://www.facebook.com/david.berlind).

@dberlind (http://twitter.com/dberlind) (https://www.facebook.com/david.berlind) (https://plus.google.com/+DavidBerlind) (http://www.linkedin.com/in/davidberlind/) (mailto:david.berlind@programmableweb.com)

‹ How the API Principles of Abstraction Benefit API Providers (/news/how-api-principles-abstraction-benefit-api-providers/analysis/2015/12/03)

Daily API RoundUp: Twilio, UserBob, Hyperwallet, cuteRadio, LeadFinch, Amitad ›

Find out more about... API Education (/category/api-education)

COMMENTS (0)

POST COMMENT

IN THIS SERIES

What Are APIs and How Do They Work? (https://www.programmableweb.com/api-university/what-are-apis-and-how-do-they-work)

Intro: APIs Are Like User Interfaces--Just With Different Users in Mind (/news/apis-are-user-interfaces-just-different-users-mind/analysis/2015/12/03)

Reading 6

RESTful Web APIs excerpt

Services for a Changing World

RESTful Web APIs

Leonard Richardson &
Mike Amundsen
Foreword by Sam Ruby

O'REILLY®

RESTful Web APIs

The popularity of REST in recent years has led to tremendous growth in almost-RESTful APIs that miss out on many of the architecture's benefits. With this practical guide, you'll learn what it takes to design usable REST APIs that evolve over time. By focusing on solutions that cross a variety of domains, this book shows you how to create powerful and secure applications, using the tools designed for the world's most successful distributed computing system: the World Wide Web.

You'll explore the concepts behind REST, learn different strategies for creating hypermedia-based APIs, and then put everything together with a step-by-step guide to designing a RESTful web API.

- Examine API design strategies, including the collection pattern and pure hypermedia
- Understand how hypermedia ties representations together into a coherent API
- Discover how XMDP and ALPS profile formats can help you meet the web API "semantic challenge"
- Learn close to two-dozen standardized hypermedia data formats
- Apply best practices for using HTTP in API implementations
- Create web APIs with the JSON-LD standard and other Linked Data approaches
- Understand the CoAP protocol for using REST in embedded systems

"A terrific book! RESTful Web APIs covers the most important trends and practices in APIs today."

—John Musser
founder of ProgrammableWeb

Leonard Richardson, author of O'Reilly's *Ruby Cookbook*, has created several open source libraries, including Beautiful Soup.

Mike Amundsen has more than a dozen books to his credit, including *Building Hypermedia APIs with HTML5 and Node* (O'Reilly).

Sam Ruby is a co-chair of the W3C HTML Working Group and a Senior Technical Staff Member in the Emerging Technologies Group of IBM.

US $34.99 CAN $36.99
ISBN: 978-1-449-35806-8

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

CHAPTER 1
Surfing the Web

The World Wide Web became popular because ordinary people can use it to do really useful things with minimal training. But behind the scenes, the Web is also a powerful platform for distributed computing.

The principles that make the Web usable by ordinary people also work when the "user" is an automated software agent. A piece of software designed to transfer money between bank accounts (or carry out any other real-world task) can accomplish the task using the same basic technologies a human being would use.

As far as this book is concerned, the Web is based on three technologies: the URL naming convention, the HTTP protocol, and the HTML document format. URL and HTTP are simple, but to apply them to distributed programming you must understand them in more detail than the average web developer does. The first few chapters of this book are dedicated to giving you this understanding.

The story of HTML is a little more complicated. In the world of web APIs, there are dozens of data formats competing to take the place of HTML. An exploration of these formats will take up several chapters of this book, starting in Chapter 5. For now, I want to focus on URL and HTTP, and use HTML solely as an example.

I'm going to start off by telling a simple story about the World Wide Web, as a way of explaining the principles behind its design and the reasons for its success. The story needs to be simple because although you're certainly familiar with the Web, you might not have heard of the concepts that make it work. I want you to have a simple, concrete example to fall back on if you ever get confused about terminology like "hypermedia as the engine of application state."

Let's get started.

Episode 1: The Billboard

One day Alice is walking around town and she sees a billboard (Figure 1-1).

Figure 1-1. The billboard

(By the way, this fictional billboard advertises a real website that I designed for this book. You can try it out yourself.)

Alice is old enough to remember the mid-1990s, so she recalls the public's reaction when URLs started showing up on billboards. At first, people made fun of these weird-looking strings. It wasn't clear what "http://" or "youtypeitwepostit.com" meant. But 20 years later, everyone knows what to do with a URL: you type it into the address bar of your web browser and hit Enter.

And that's what Alice does: she pulls out her mobile phone and puts *http://www.youtypeitwepostit.com/* in her browser's address bar. The first episode of our story ends on a cliffhanger: what's at the other end of that URL?

Resources and Representations

Sorry for interrupting the story, but I need to introduce some basic terminology. Alice's web browser is about to send an HTTP request to a web server—specifically, to the URL *http://www.youtypeitwepostit.com/*. One web server may host many different URLs, and each URL grants access to a different bit of the data on the server.

We say that a URL is the URL of some thing: a product, a user, the home page. The technical term for the thing named by a URL is *resource*.

The URL *http://www.youtypeitwepostit.com/* identifies a resource—probably the home page of the website advertised on the billboard. But you won't know for sure until we resume the story and Alice's web browser sends the HTTP request.

When a web browser sends an HTTP request for a resource, the server sends a document in response (usually an HTML document, but sometimes a binary image or something else). Whatever document the server sends, we call that document a *representation* of the resource.

So each URL identifies a resource. When a client makes an HTTP request to a URL, it gets a representation of the underlying resource. The client never sees a resource directly.

I'll talk a lot more about resources and representations in Chapter 3. Right now I just want to use the terms resource and representation to discuss the principle of addressability, to which I'll now turn.

Addressability

A URL identifies one and only one resource. If a website has two conceptually different things on it, we expect the site to treat them as two resources with different URLs. We get frustrated when a website violates this rule. Websites for restaurants are especially bad about this. Frequently, the whole site is buried inside a Flash interface and there's no URL that points to the menu or to the map that shows where the restaurant is located—things we would like to talk about on their own.

The principle of addressability just says that every resource should have its own URL. If something is important to your application, it should have a unique name, a URL, so that you and your users can refer to it unambiguously.

Episode 2: The Home Page

Back to our story. When Alice enters the URL from the billboard into her browser's address bar, it sends an HTTP request over the Internet to the web server at *http://www.youtypeitwepostit.com/*:

```
GET / HTTP/1.1
Host: www.youtypeitwepostit.com
```

The web server handles this request (neither Alice nor her web browser need to know how) and sends a response:

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <div>
          <h1>You type it, we post it!</h1>
```

HTTP sessions last for one request. The client sends a request, and the server responds. This means Alice could turn her phone off overnight, and when her browser restored the page from its internal cache, she could click on one of the two links on this page and it would still work. (Compare this to an SSH session, which is terminated if you turn your computer off.)

Alice could leave this web page open in her phone for six months, and when she finally clicks on a link, the web server would respond as if she'd only waited a few seconds. The web server isn't sitting up late at night worrying about Alice. When she's not making an HTTP request, the server doesn't know Alice exists.

This principle is sometimes called statelessness. I think this is a confusing term because the client and the server in this system both keep state; they just keep different *kinds* of state. The term "statelessness" is getting at the fact that the *server* doesn't care what state the *client* is in. (I'll talk more about the different kinds of state in the following sections.)

Self-Descriptive Messages

It's clear from looking at the HTML that this site is more than just a home page. The markup for the home page contains two links: one to the relative URL */about* (i.e., to *http://www.youtypeitwepostit.com/about*) and one to */messages* (i.e., *http://www.youtypeitwepostit.com/messages*). At first Alice only knew one URL—the URL to the home page—but now she knows three. The server is slowly revealing its structure to her.

We can draw a map of the website so far (Figure 1-3), as revealed to Alice by the server.

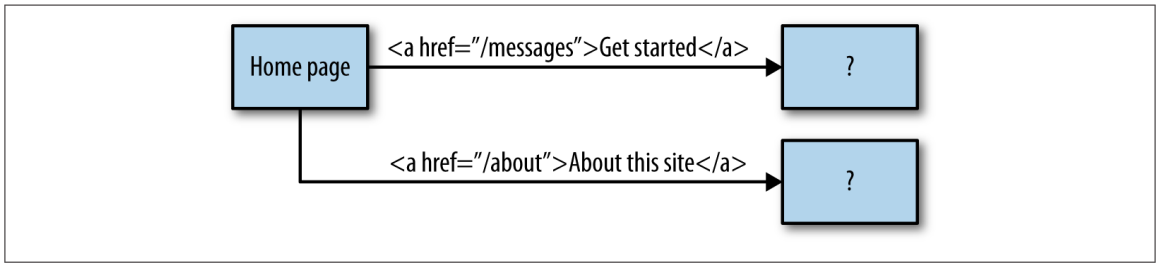

Figure 1-3. A map of the website

What's on the other end of the */messages* and */about* links? The only way to be sure is to follow them and find out. But Alice can look at the HTML markup, or her browser's graphical rendering of the markup, and make an educated guess. The link with the text "About this site" probably goes to a page talking about the site. That's nice, but the link with the text "Get started" is probably the one that gets her closer to actually posting a message.

When you request a web page, the HTML document you receive doesn't just give you the immediate information you asked for. The document also helps you answer the question of what to do next.

Episode 3: The Link

After reading the home page, Alice decides to give this site a try. She clicks the link that says "Get started." Of course, whenever you click a link in your web browser, you're telling your web browser to make an HTTP request.

The code for the link Alice clicked on looks like this:

```
<a href="/messages">Get started</a>
```

So her browser makes this HTTP request to the same server as before:

```
GET /messages HTTP/1.1
Host: www.youtypeitwepostit.com
```

That GET in the request is an *HTTP method*, also known as an *HTTP verb*. The HTTP method is the client's way of telling the server what it wants to do to a resource. "GET" is the most common HTTP method. It means "give me a representation of this resource." For a web browser, GET is the default. When you follow a link or type a URL into the address bar, your browser sends a GET request.

The server handles this particular GET request by sending a representation of */messages*:

```
HTTP/1.1 200 OK
Content-type: text/html
...

<!DOCTYPE html>
<html>
    <head>
        <title>Messages</title>
    </head>
    <body>
      <div>
        <h1>Messages</h1>

        <p>
          Enter your message below:
        </p>

        <form action="http://youtypeitwepostit.com/messages" method="post">
            <input type="text" name="message" value="" required="true"
                   maxlength="6"/>
            <input type="submit" value="Post" />
        </form>

        <div>
```

```
      <p>
        Here are some other messages, too:
      </p>
      <ul>
        <li><a href="/messages/32740753167308867">Later</a></li>
        <li><a href="/messages/7534227794967592">Hello</a></li>
      </ul>
    </div>

    <p class="links">
      <a href="http://youtypeitwepostit.com/">Home</a>
    </p>

  </div>
 </body>
</html>
```

As before, Alice's browser renders the HTML graphically (Figure 1-4).

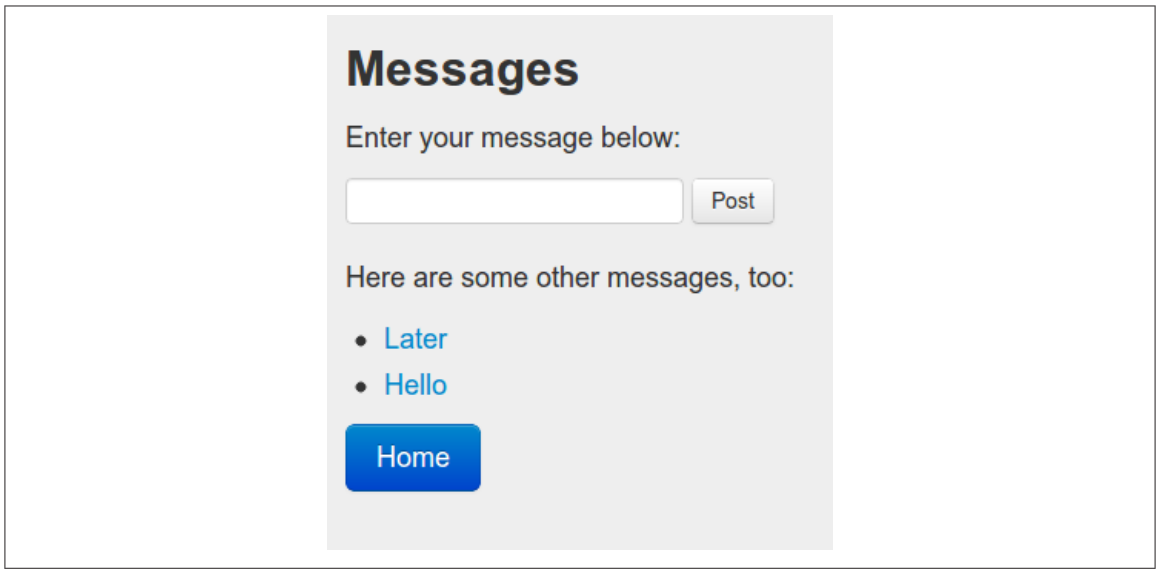

Figure 1-4. You Type It… "Get started" page

When Alice looks at the graphical rendering, she sees that this page is a list of messages other people have published on the site. Right at the top there's an inviting text box and a Post button.

Now we've revealed a little more about how the server works. Figure 1-5 shows an updated map of the site, as seen by Alice's browser.

Episode 4: The Form and the Redirect

Back to our story. Alice is tempted by the form on the microblogging site. She types in "Test" and clicks the Post button.:

Again, Alice's browser makes an HTTP request:

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-type: application/x-www-form-urlencoded

message=Test&submit=Post
```

And the server responds with the following:

```
HTTP/1.1 303 See Other
Content-type: text/html
Location: http://www.youtypeitwepostit.com/messages/5266722824890167
```

When Alice's browser made its two GET requests, the server sent the HTTP status code 200 ("OK") and provided an HTML document for Alice's browser to render. There's no HTML document here, but the server did provide a link to another URL, in the `Location` header—and here, the status code at the beginning of the response is 303 ("See Other"), not 200 ("OK").

Status code 303 tells Alice's browser to *automatically* make a fourth HTTP request, to the URL given in the `Location` header. Without asking Alice's permission, her browser does just that:

```
GET /messages/5266722824890167 HTTP/1.1
```

This time, the browser responds with 200 ("OK") and an HTML document:

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
    <head>
        <title>Message</title>
    </head>
    <body>
        <div>
            <h2>Message</h2>
            <dl>
                <dt>ID</dt><dd>2181852539069950</dd>
                <dt>DATE</dt><dd>2014-03-28T21:51:08Z</dd>
                <dt>MSG</dt><dd>Test</dd>
            </dl>
            <p class="links">
                <a href="http://www.youtypeitwepostit.com/">Home</a>
            </p>
```

```
        </div>
    </body>
</html>
```

Alice's browser displays this document graphically (Figure 1-6), and, finally, goes back to waiting for Alice's input.

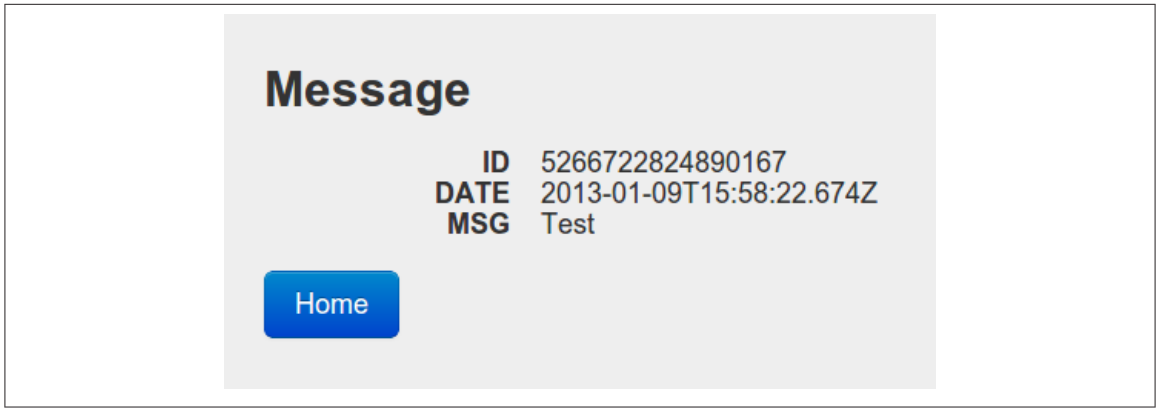

Figure 1-6. You Type It… posted message

I'm sure you've encountered HTTP redirects before, but HTTP is full of small features like this, and some may be new to you. There are many ways for the server to tell the client to handle a response differently, and ways for the client to attach conditions or extra features to its request. A big part of API design is the proper use of these features. Chapter 11 covers the features of HTTP that are most important to web APIs, and Appendix A and Appendix B provide supplementary information on this topic.

By looking at the graphical rendering, Alice sees that her message ("Test") is now a fully fledged post on YouTypeItWePostIt.com. Our story ends here—Alice has accomplished her goal of trying out the microblogging site. But there's a lot to be learned from these four simple interactions.

Application State

Figure 1-7 is a state diagram that shows Alice's entire adventure from the perspective of her web browser.

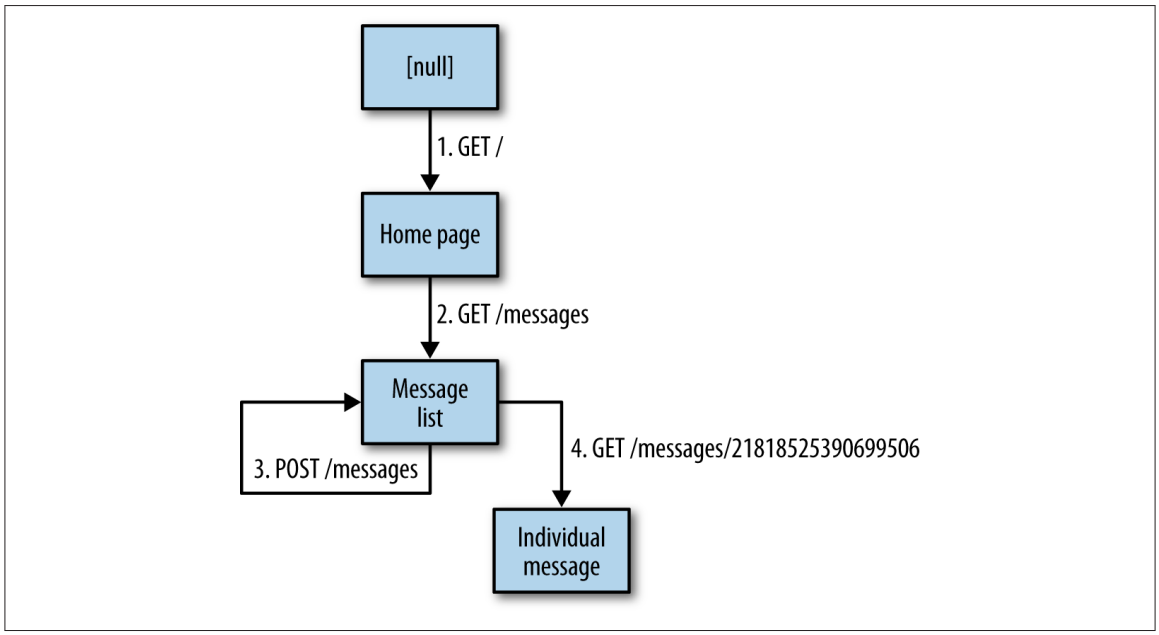

Figure 1-7. Alice's adventure: the client's perspective

When Alice started up the browser on her phone, it didn't have any particular page loaded. It was an empty slate. Then Alice typed in a URL and a GET request took the browser to the site's home page. Alice clicked a link, and a second GET request took the browser to the list of messages. She submitted a form, which caused a third request (a POST request). The response to that was an HTTP redirect, which Alice's browser made automatically. Alice's browser ended up at a web page describing the message Alice had just created.

Every state in this diagram corresponds to a particular page (or to no page at all) being open in Alice's browser window. In REST terms, we call this bit of information—which page are you on?—the *application state*.

When you surf the Web, every transition from one application state to another corresponds to a link you decided to follow or a form you decided to fill out. Not all transitions are available from all states. Alice can't make her POST request directly from the home page, because the home page doesn't feature the form that allows her browser to construct the POST request.

Resource State

Figure 1-8 is a state diagram showing Alice's adventure from the perspective of the web server.

Because HTTP sessions are so short, the server doesn't know anything about a client's application state. The client has no direct control over resource state—all that stuff is kept on the server. And yet, the Web works. It works through REST—representational state transfer.

Application state is kept on the client, but the server can manipulate it by sending representations—HTML documents, in this case—that describe the possible state transitions. Resource state is kept on the server, but the client can manipulate it by sending the server a representation—an HTML form submission, in this case—describing the desired new state.

Connectedness

In the story, Alice made four HTTP requests to YouTypeItWePostIt.com, and she got three HTML documents in return. Although Alice didn't follow every single link in those documents, we can use those links to build a rough map of the website from the client's perspective (Figure 1-9).

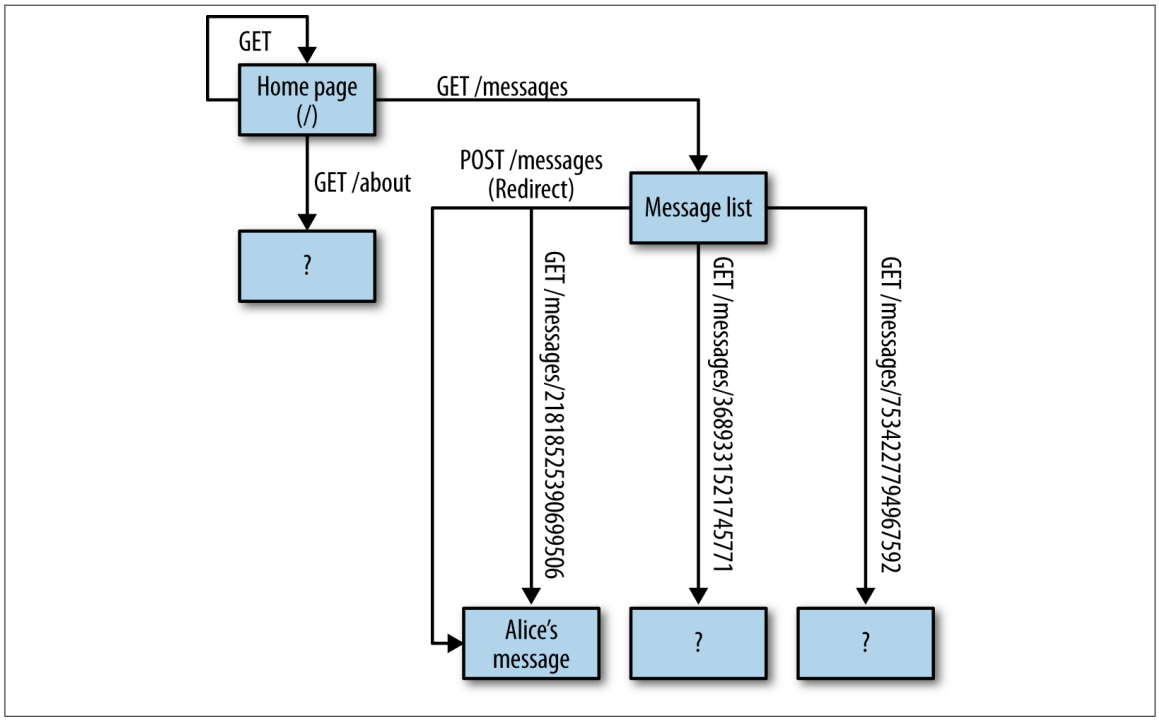

Figure 1-9. What the client saw

This is a web of HTML pages. The strands of the web are the HTML `<a>` tags and `<form>` tags, each describing a GET or POST HTTP request Alice might decide to make. I call this the principle of *connectedness*: each web page tells you how to get to the adjoining pages.

The Web as a whole works on the principle of connectedness, which is better known as "hypermedia as the engine of application state," sometimes abbreviated HATEOAS. I prefer "connectedness" or "the hypermedia constraint," because "hypermedia as the engine of application state" sounds intimidating. But at this point, you should have no reason to find it intimidating. You know what application state is—it's which web page a client is on. *Hypermedia* is the general term for things like HTML links and forms: the techniques a server uses to explain to a client what it can do next.

To say that hypermedia is the engine of application state is to say that we all navigate the Web by filling out forms and following links.

The Web Is Something Special

Alice's story doesn't seem that exciting. because the World Wide Web has been the dominant Internet application for the past 20 years. But back in the 1990s, this was a very exciting story. If you compare the World Wide Web to its early competitors, you'll see the difference.

The Gopher protocol (defined in RFC 1436) looks a lot like HTTP, but it lacks addressability. There is no succinct way to identify a specific document in Gopherspace. At least there wasn't until the World Wide Web took pity on Gopherspace and released the URL standard (first defined in RFC 1738), which provides a *gopher://* URL scheme that works just like *http://*.

FTP, a popular pre-Web protocol for file transfer (defined in RFC 959), also lacks addressability. Until RFC 1738 came along with its *ftp://* URL scheme, there simply was no machine-readable way to point to a file on an FTP server. You had to use English prose to explain where the file was. It took the brainpower of a human being just to locate a file on a server. What a waste!

FTP also featured long-lived sessions. A casual user could log on to an FTP server and tie up one of the server's TCP connections indefinitely. By contrast, even a "persistent" HTTP connection shouldn't tie up a TCP connection for longer than 30 seconds.

The 1990s saw a lot of Internet protocols for searching different kinds of archives and databases—protocols like Archie, Veronica, Jughead, WAIS, and Prospero. But it turns out we don't need all those protocols. We just need to be able to send GET requests to different kinds of websites. All these protocols died out or were replaced by websites. Their complex protocol-specific rules were folded into the uniformity of HTTP GET.

Once the Web took over, it became a lot more difficult to justify creating a new application protocol. Why create a new tool that only techies will understand, when you can put up a website that anyone can use? All successful post-Web protocols do something the Web can't do: peer-to-peer protocols like BitTorrent and real-time protocols like SSH. For most purposes, HTTP is good enough.

The unprecedented flexibility of the Web comes from the principles of REST. In the 1990s, we discovered that the Web works better than its competition. In 2000, Roy T. Fielding's Ph.D dissertation[1] explained why this is, coining the term "REST" in the process.

Web APIs Lag Behind the Web

The Fielding dissertation also explains a lot about the problems of web APIs in the 2010s. The simple website I just walked you through is much more sophisticated than most currently deployed web APIs—even self-proclaimed REST APIs. If you've ever designed a web API, or written a client for one, you've probably encountered some of these problems:

- Web APIs frequently have human-readable documentation that explains how to construct URLs for all the different resources. This is like writing English prose explaining how to find a particular file on an FTP server. If websites did this, no one would bother to use the Web.

  Instead of telling you what URLs to type in, websites embed URLs in `<a>` tags and `<form>` tags—hypermedia controls that you can activate by clicking a link or a button.

  In REST terms, putting information about URL construction in separate human-readable documents violates the principles of connectedness and self-descriptive messages.

- Lots of websites have help docs, but when was the last time you used them? Unless there's a serious problem (you bought something and it was never delivered), it's easier to click around and figure out how the site works by exploring the connected, self-descriptive HTML documents it sends you.

  Today's APIs present their resources in a big menu of options instead of an interconnected web. This makes it difficult to see what one resource has to do with another.

- Integrating with a new API inevitably requires writing custom software, or installing a one-off library written by someone else. But you don't need to write custom software to use a new website. You see a URL on a billboard and plug it into your web browser—the same client you use for every other website in the world.

  We'll never get to the point where a single API client can understand every API in the world. But today's clients contain a lot of code that really ought to be refactored

1. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

CHAPTER 2
A Simple API

In Chapter 1, I showed off a very simple microblogging website located at *http://www.youtypeitwepostit.com/*. As it happens, I've also designed a programmable API for this website. You can see it live at *http://www.youtypeitwepostit.com/api/*.

The ideal API would have the same characteristics that make the World Wide Web easy to use. As a developer, you would be able to figure out how to use it, starting with nothing but a URL you saw on a billboard.

Let's spin out that fantasy to see how it would work. First, you would have your programmable client make a GET request for the billboard URL—the equivalent to entering that URL into your web browser's address bar. Your client would take over from there, examining the response to see what the available options are. It would follow links (not necessarily HTML links), fill out forms (not necessarily HTML forms), and eventually accomplish the task you set out for it.

This book is not going to get us all the way to that goal. There are problems I can't solve with a book: problems surrounding the absence of standards, problems with the current level of tool support, and the brute fact that computers just aren't as smart as human beings. But we can get a long way toward that goal—a lot further than you may think.

As I said, there is a real microblogging API at *http://www.youtypeitwepostit.com/api/*. If you're feeling adventurous, go ahead and write some code to do something with that API. See how much you can figure out, knowing nothing but that URL. You've done this before with websites: all you knew was the home page URL, and you figured it out. How far can you get with an API?

If you're not feeling adventurous or don't have much experience writing clients for web APIs (or you're reading this book in the far future, and I'm not hosting that website anymore), we'll go through it together. The first step is to get a representation of the API's home page.

HTTP GET: Your Safe Bet

If you have a URL that starts with *http://* or *https://*, and you don't know what's on the other side, the first thing to do is to issue an HTTP GET request. In REST terms, you know the URL to a resource and nothing else. You need to discover your options, and that means getting a representation of the resource. That's what HTTP GET is for.

You can write code in a programming language to make that GET request, but when doing an initial reconnaissance of an API, it's often easier to use a command-line tool like Wget. Here I use the `-S` option, which prints out the full HTTP response from the server, and the `-O -` option, which prints out the document instead of saving it to a file:

```
$ wget -S -O - http://www.youtypeitwepostit.com/api/
```

This sends an HTTP request like this to the server:

```
GET /api/ HTTP/1.1
Host: www.youtypeitwepostit.com
```

The HTTP standard says that a GET request is a request for a representation. It's not intended to change any resource state on the server. This means that if you have a URL to a resource and don't know anything more, you can always make a GET request and get a representation in return. Your GET request won't do something disastrous like delete all the data. We say that GET is a *safe* method.

It's OK for the server to change incidental things because of a GET request, like incrementing a hit counter or logging the request to a file, but that's not the *purpose* of the GET request. Nobody makes an HTTP request just to increment the hit counter.

In real life, there's no guarantee that HTTP GET is safe. Some older designs *will* force you to make an HTTP GET request if you want to delete some data. But this misfeature is pretty rare in newer designs. Most API designers now understand that clients frequently GET a URL just to see what's behind it. It's not fair to give that GET request significant side effects.

How to Read an HTTP Response

In response to my GET request, the server sends a big chunk of data that looks like this:

```
HTTP/1.1 200 OK
ETag: "f60e0978bc9c458989815b18ddad6d75"
Last-Modified: Thu, 10 Jan 2013 01:45:22 GMT
Content-Type: application/vnd.collection+json

{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",
    "items" : [
```

```
      { "href" : "http://www.youtypeitwepostit.com/api/messages/21818525390699506",
        "data": [
         { "name": "text", "value": "Test." },
         { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }
        ],
        "links": []
      },

      { "href" : "http://www.youtypeitwepostit.com/api/messages/3689331521745771",
        "data": [
         { "name": "text", "value": "Hello." },
         { "name": "date_posted", "value": "2013-04-20T12:55:59.685Z" }
        ],
        "links": []
      },

      { "href" : "http://www.youtypeitwepostit.com/api/messages/7534227794967592",
        "data": [
         { "name": "text", "value": "Pizza?" },
         { "name": "date_posted", "value": "2013-04-18T03:22:27.485Z" }
        ],
        "links": []
      }
      ]
    },

    "template": {
      "data": [
         {"prompt": "Text of message", "name": "text", "value":""}
      ]
    }
  }
```

How much can we learn from this? Well, every HTTP response can be split into three parts:

The status code, sometimes called the response code
: This is a three-digit number that summarizes how the request went. The response code is the first thing an API client sees, and it sets the tone for the rest of the response. Here, the status code was 200 (`OK`). This is the status code a client hopes for—it means that everything went fine.

  In Appendix A, I explain all of the standard HTTP response codes, as well as several useful extensions.

The entity-body, sometimes called just the body
: This is a document written in some data format, which the client is expected to understand. If you think of a GET request as a request for a representation, you can think of the entity-body as the representation (technically, the entire HTTP re-

Collection+JSON

So, this entity-body document is JSON, right? Not so fast! You can feed this document into a JSON parser without crashing the parser, but that's not what the web server wants you to do. Here's what the server said:

```
Content-Type: application/vnd.collection+json
```

That conflicts with the JSON RFC, which says a JSON document should be served as `application/json`, like this:

```
Content-Type: application/json
```

So what is this `application/vnd.collection+json` stuff? Clearly this format is *based* on JSON, since it looks like JSON and its media type has "json" in the name. But what is it, really?

If you search the web for `application/vnd.collection+json`, you'll discover that it's a media type registered for Collection+JSON.[1] When you make a GET request to *http://www.youtypeitwepostit.com/api/*, you don't get just any JSON document—you get a Collection+JSON document.

In Chapter 6, I'll talk about Collection+JSON in detail, but here's the short version. Collection+JSON is a standard for publishing a searchable list of resources over the Web. JSON puts constraints on plain text, and Collection+JSON puts constraints on JSON. A server can't serve just any JSON document as `application/vnd.collection+json`. It can only serve a JSON object:

```
{}
```

But not just any object. The object has to have a property called `collection`, which maps to another object:

```
{"collection": {}}
```

The "collection" object ought to have a property called `items` that maps to a list:

```
{"collection": {"items": []}}
```

The items in the "items" list need to be objects:

```
{"collection": {"items": [{}, {}, {}]}}
```

And on and on, constraint after constraint. Eventually you get the highly formatted document you just saw, which starts out like this:

```
{ "collection":
  {
    "version" : "1.0",
```

1. Collection+JSON is a personal standard defined at this page.

```
      "href" : "http://www.youtypeitwepostit.com/api/",
      "items" : [

      { "href" : "http://www.youtypeitwepostit.com/api/messages/21818525390699506",
        "data": [
         { "name": "text", "value": "Test." },
         { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }
        ],
        "links": []
      },
...
}
```

Look at the document as a whole, and the purpose of all these constraints becomes clear. Collection+JSON is a way of serving lists—not lists of data structures, which you can do with normal JSON, but lists that describe HTTP resources.

The `collection` object has an `href` property, and its value is a JSON string. But it's not just any string—it's the URL I just sent a GET request to:

```
{ "collection":
  {
    "href" : "http://www.youtypeitwepostit.com/api/"
  }
}
```

The Collection+JSON standard defines this string as "the address used to retrieve a representation of the document" (in other words, it's the URL of the `collection` resource). Each object inside the collection's `items` list has its own `href` property, and each value is a string containing a URL, like *http://www.youtypeitwepostit.com/api/messages/21818525390699506* (in other words, each item in the list represents an HTTP resource with its own URL).

A document that doesn't follow these rules isn't a Collection+JSON document: it's just some JSON. By allowing yourself to be bound by Collection+JSON's constraints, you gain the ability to talk about concepts like resources and URLs. These concepts are not defined in JSON, which can only talk about simple things like strings and lists.

Writing to an API

How would I use the API to publish a message to the microblog? Here's what the Collection+JSON specification has to say:

> To create a new item in the collection, the client first uses the `template` object to compose a valid `item` representation and then uses HTTP POST to send that representation to the server for processing.

That's not exactly a step-by-step description, but it points toward the answer. Collection +JSON works along the same lines as HTML. The server provides you with some kind

of form (the `template`), which you fill out to create a document. Then you send that document to the server with a POST request.

Again, Chapter 6 covers Collection+JSON in detail, so here's the quick version. Look at the big object I showed you earlier. Its `template` property is the "`template` object" mentioned in the Collection+JSON specification:

```
{
 ...
 "template": {
   "data": [
      {"prompt": "Text of message", "name": "text", "value":""}
   ]
}
```

To fill out the template, I replace the empty string under `value` with the string I want to publish:

```
 { "template":
   {
     "data": [
         {"prompt": "Text of the message", "name": "text", "value": "Squid!"}
     ]
   }
 }
```

I then send the filled-out template as part of an HTTP POST request:

```
POST /api/ HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/vnd.collection+json

{ "template":
 {
  "data": [
   {"prompt": "Text of the message", "name": "text", "value": "Squid!"}
  ]
 }
}
```

(Note that my request's `Content-Type` is `application/vnd.collection+json`. This filled-out template is a valid Collection+JSON document all on its own.)

The server responds:

```
HTTP/1.1 201 Created
Location: http://www.youtypeitwepostit.com/api/47210977342911065
```

The 201 response code (`Created`) is a little more specific than 200 (`OK`); it means that everything is OK *and* that a new resource was created in response to my request. The `Location` header gives the URL to the newborn resource.

```
{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/47210977342911065",
    "items" : [

    { "href" : "http://www.youtypeitwepostit.com/api/messages/47210977342911065",
      "data": [
       { "name": "date_posted", "value": "2014-04-20T20:15:32.858Z" },
       { "name": "text", "value": "Squid!" }
      ],
      "links": []
    }
    ]
  }
}
```

This individual microblog post is represented as a full `application/vnd.collection+json` document. It's a `collection` with an `items` list that only contains one item. The filled-out template was also a valid `application/vnd.collection+json` document, even though it didn't use the `collection` property at all.

This is a convenience feature of Collection+JSON. Almost everything in the document is optional. It means you don't have to write different parsers to handle different types of documents. Collection+JSON uses the same JSON format to represent lists of items, individual items, filled-out templates, and search results.

Liberated by Constraints

One counterintuitive lesson of RESTful design is that constraints can be liberating. The safety constraint of HTTP's GET method is a good example. Thanks to the safety constraint, you know that if you don't know what to do with a URL, you can always GET it and look at the representation. Even if that doesn't help, nothing terrible will happen just because you made a GET request. That's a liberating promise, and it's only possible because of a very severe constraint on the server side.

If the server sends you a plain text document that says `9`, you have no way to know if it's supposed to be the number nine or the string "9". But if you get a JSON document that says `9`, you know it's a number. The JSON standard constrains the meaning of the document, and that makes it possible for client and server to have a meaningful conversation.

Over the past few years, hundreds of companies have gone through this general line of thinking:

1. We need an API.
2. We'll use JSON as the document format.

3. We'll use JSON to publish lists of things.

All three of these are good ideas, but they don't say much about what the API should look like. The end result is hundreds of APIs that are superficially similar (they all use JSON to publish lists of things!) but completely incompatible. Learning one API doesn't help a client learn the next one.

This is a sign that more constraints are necessary. The Collection+JSON standard provides some more constraints. If I'd come up with my own custom API design instead of using Collection+JSON, an individual item in my list might have looked like this:

```
{
 "self_link": "http://www.youtypeitwepostit.com/api/messages/47210977342911065",
 "date": "2014-04-20T20:15:32.858Z",
 "text": "Squid!"
}
```

Instead, because I followed the Collection+JSON constraints, an individual item looks like this:

```
{ "href" : "http://www.youtypeitwepostit.com/api/messages/1xe5",
  "data": [
   { "name": "date_posted", "value": "2014-04-20T20:15:32.858Z" },
   { "name": "text", "value": "Squid!" }
  ],
  "links": []
}
```

The custom design is certainly more compact, but that's not very important—JSON compresses very well. In exchange for this less compact representation, I get a number of useful features:

- I don't have to tell all my users that the value of `href` is a URL, and I don't have to explain what it's the URL of. The Collection+JSON standard says that an item's `href` contains the URL to the item.
- I don't have to write a separate human-readable document explaining to my users that `text` is the text of the message. That information goes where it's actually needed —in the template you fill out to post a new message:

```
"template": {
  "data": [
     {"prompt": "Text of the message", "name": "text", "value": null}
  ]
}
```

- Any library that understands `application/vnd.collection+json` automatically knows how to use my API. If I came up with a custom design, I'd have to write

brand new client code based on nothing but a JSON parser and an HTTP library, or ask all my users to write that code themselves.

By submitting to the Collection+JSON constraints, I free myself from having to write a whole lot of documentation and code, and I free my users from having to learn yet another custom API.

Application Semantics Create the Semantic Gap

Of course, the Collection+JSON constraints don't constrain everything. Collection +JSON doesn't specify that the items in a collection should be microblog posts with a `date_posted` and a `text`. I made that part up, because I wanted to design a simple microblogging example for this book. If I'd chosen to do a "recipe book" example, I could still use Collection+JSON, but the items would have data fields like `ingredients` and `preparation_time`.

I'm going to call these extra bits of design the *application semantics*, because they vary from one application to another. Application semantics are the cause of the semantic gap I mentioned in Chapter 1.

If I were designing a real microblogging API, I'd come up with application semantics more complicated than just `text` and `date_posted`. That's fine, on its own. But there are currently dozens of companies designing microblogging APIs, coming up with dozens of designs that feature mutually incompatible application semantics, creating dozens of distinct semantic gaps. All of these companies are doing the same thing in different ways. Their users have to write different software clients to accomplish the same task.

The fact that Collection+JSON doesn't solve this problem doesn't mean there's no point to using Collection+JSON. Compatibility is a matter of degree. We took a big step toward compatibility in the 1990s when we stopped inventing custom Internet protocols and standardized on HTTP. If we all agreed to serve JSON documents, that might not be a good idea technically, but it would narrow the semantic gap. Standardizing on Collection+JSON would narrow it even more.

If the publishers of microblogging APIs got together and agreed to use a common set of application semantics, the semantic gap for microblogging would disappear almost entirely. (This would be a *profile*, and I'll cover this idea in Chapter 8.) The more constraints we share and the more compatible our designs, the smaller the semantic gap and the more our users benefit.

Maybe you don't *want* your API to be interoperable with your competitor's APIs, but there are better ways to differentiate yourself than by artificially widening the semantic gap. My goal for this book is to get you focused on the parts of your API that have something new to offer, in the spots where a semantic gap exists because no one else has ever taken that path.

Reading 7

APIs: A Strategy Guide excerpt

Creating Channels with Application Programming Interfaces

APIs

A Strategy Guide

Daniel Jacobson,
Greg Brail & Dan Woods

O'REILLY®

APIs: A Strategy Guide

Programmers used to be the only people excited about APIs, but now a growing number of companies see them as a hot new product channel. This concise guide describes the tremendous business potential of APIs, and demonstrates how you can use them to provide valuable services to clients, partners, or the public via the Internet. You'll learn all the steps necessary for building a cohesive API business strategy from experts in the trenches.

Facebook and Twitter APIs continue to be extremely successful, and many other companies find that API demand greatly exceeds website traffic. This book offers executives, business development teams, and other key players a complete roadmap for creating a viable API product.

- Learn about the rise of APIs and why your business might need one
- Understand the roles of asset owners, providers, and developers in the API value chain
- Build strategies for designing, implementing, and marketing your product
- Devise an effective process for security and user management
- Address legal issues, such as rights management and terms of use
- Manage traffic and user experience with a reliable operating model
- Determine the metrics you need to measure your API's success

Purchase the ebook edition of this O'Reilly title at oreilly.com and get free updates for the life of the edition. Our ebooks are optimized for several electronic formats, including PDF, EPUB, Mobi, APK, and DAISY—all DRM-free.

Twitter: @oreillymedia
facebook.com/oreilly

US $24.99 CAN $26.99
ISBN: 978-1-449-30892-6

O'REILLY®
oreilly.com

APIs: A Strategy Guide

Daniel Jacobson, Greg Brail, and Dan Woods

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

CHAPTER 1
The API Opportunity

APIs are a big deal and they are getting bigger. Pioneering companies such as Google, Facebook, Apple, and Twitter have exposed amazing technological solutions to the public, transforming existing businesses and creating new industries. Central to these companies' successes are the APIs that link people and their computing devices to the underlying platforms that power each business and that tie these companies together behind the scenes.

The world is already changing. Consider the following examples:

- Salesforce.com built a large, rich partner ecosystem by opening core services for partners to innovate and extend. Today, more traffic comes through the Salesforce API than through its website. As of mid-2011 more than 60 percent of the traffic to Salesforce.com comes through APIs.
- Amazon opened its core computing infrastructure as Amazon Web Services (AWS), accessed via number of APIs, and now serves more bandwidth through AWS than through all of its global storefronts combined.
- Twitter is the most visible example of a business almost entirely based on an API and an ecosystem of developer applications.
- Netflix has completely reinvented how we consume movies and TV shows with streaming to hundreds of different devices, upending not just the video rental industry but also impacting large adjacent markets such as cable TV. APIs allow Netflix to support a multitude of devices in an affordable manner.
- NPR has infused its API into the engineering culture of the digital media division. The API drives the website, mobile apps, and all other forms of distribution and syndication for the company. The API has also transformed the company's relationship with its member stations and the way that NPR shares content with them.

Now consider these industry trends:

- Smartphone sales passed new PC sales in early 2011, and Morgan Stanley predicts that by the end of 2012, there will be more connected mobile devices in the world than PCs.

- CTIA (the wireless industry association) has determined that there are already more wireless devices in the United States than people.
- Analysts are competing to predict how many mobile devices will exist in the future. The GSMA (another wireless industry association) predicts that there will be 20 billion connected mobile devices by 2020, and Ericsson CEO Hans Vestberg predicts 50 billion. Meanwhile, Marthin De Beer, Senior Vice President of Cisco's Emerging Technologies Group, projects that count to be over a trillion by 2020.
- Cisco predicts that while Internet traffic originated by PCs will continue to grow at 33 percent per year, traffic originated by non-PC devices will more than double each year by 2015.
- Facebook accounts for over 25 percent of total Internet page views at this writing, and APIs drive both the Facebook products and its ecosystem.
- Over 30 percent of Internet traffic during US prime-time hours comes from Netflix streaming, which is delivered and managed via APIs.

These statistics point not only to an explosion of overall Internet traffic, but also to a huge shift in the distribution of this traffic towards apps and devices. Looking at these accelerating trends, it's very easy to imagine that APIs will likely power most of your Internet traffic within a few years.

Why We Wrote This Book

As authors, we are coming at this topic fresh from our experiences in the trenches. Daniel Jacobson led development of the NPR content management system and the API that draws from that system. The NPR API is now the centerpiece of NPR's digital distribution strategy, transforming NPR's ability to reach its audience on a wide range of platforms.

Today, Daniel leads the development of APIs at Netflix, whose API strategy is in the critical path of the Netflix streaming service. Netflix's ability to provide rich video experiences on hundreds of devices is powered by this service and has dramatically increased the rate at which new implementations can be built and delivered to new devices. Through this program, Netflix's user base has grown tremendously, resulting in API growth from under one billion requests per month to more than one billion requests per day, in one year.

Greg Brail writes based on his work as CTO of Apigee, where he has helped implement dozens of API programs and been exposed to many more. In this role he is also able to draw from the collective wisdom of the Apigee team, who has met hundreds of developers and also learned from hundreds of enterprise API programs.

We wrote this book to help people understand the potential of APIs. Additionally, we want you to go into creating an API with your eyes wide open. This book is not a

programming manual but a best practices manual. You need to understand both the opportunity and the tactical issues related to creating an API.

This book will also introduce business executives and technologists to the land of API opportunity. To be sure, the world of APIs involves lots of technology, but what often gets lost in the shuffle is the business impact of APIs. This book is about how to think about APIs from a business perspective and how APIs can have a positive impact on your business.

We also want to educate you on what you're getting yourself into when you decide to develop an API. What are the implications of offering an API, not just from a technology standpoint but also in terms of business strategy, legal and rights considerations, and relationships with partners?

What we are going to demonstrate is that APIs are having a profound impact on the world of business—and that the time to act is now.

Unlike many other discussions of APIs that exclusively look at the way that large Internet-based companies use APIs publicly, this book also emphasizes the private use of APIs, which we believe has an even greater impact than many of the more prolific public API programs you often read about.

As authors, we must keep one foot in the world of technology and one foot in the world of business. To that end, we hope to educate creative executives and technologists about how to put APIs to work in the context of their own businesses.

In this book, we'll talk about:

- The business opportunity for APIs
- Examples of companies using APIs to transform their business and in some cases their industries
- Business models being used for APIs
- What an API value chain looks like and how to enable the different pieces of that value chain
- Considerations for crafting your API strategy and responding to concerns and objections
- Issues around API design, especially how to make adoption easy for developers
- What to do about API security, including coverage of OAuth
- The legal implications of running an API business, including privacy and data rights
- Considerations for operating your API at scale
- How to think about metrics and measuring your API program
- Engaging developers and building community to drive adoption of your API

In summary, this book offers a roadmap for using APIs to transform your business.

Our Working Definition of an API

Technical definition: An API is a way for two computer applications to talk to each other over a network (predominantly the Internet) using a common language that they both understand.

APIs follow a specification, meaning:

- The API provider describes exactly what functionality the API will offer
- The API provider describes when the functionality will be available and when it might change in an incompatible way
- The API provider may outline additional technical constraints within the API, such as rate limits that control how many times a particular application or end user is allowed to use the API in a given hour, day, or month
- The API provider may outline additional legal or business constraints when using the API, such as branding limitations, types of use, and so on
- Developers agree to use the API as described, to use only the APIs that are described, and to follow the rules set out by the API provider

In addition, the API provider may offer other tools, such as:

- Mechanisms to access the API and understand its terms of use
- Documentation to aid in understanding the API
- Resources such as example programs and developer communities to support those using the API
- Operational information about the health of the API and how much use it is getting

Remember that the structure of the API is part of the contract. The contract is binding, and it cannot be changed casually.

You should treat an API like a software product, taking into account versioning, backward compatibility, and ramp-up time to accommodate any new functionality. There should be a balance between supporting your existing base while at the same time keeping up with necessary changes so that your API grows with your business and follows its planned evolution.

This does not mean that the API can never change. On the contrary, an API is an online product that can change almost constantly to meet the needs of the business, or to serve the current traffic load in the most efficient way. But these are changes to the implementation, not to the interface. When done right, the implementation of an API can change on a daily basis, or even more often, while the interface remains consistent.

. . . But APIs and Websites Have a Lot in Common

APIs, like websites, are expected to be available 24/7, 365 days a year. Developers, like website users, do not have much patience for "scheduled downtime" every Saturday morning. All of this can create a challenge for building an API on an existing enterprise technology infrastructure, using systems that may have been designed with an "end of day" cycle, after which they are shut down until the next day (such as banking systems).

Successful websites can, and are, updated continuously to change the design and tweak features. This is possible because websites are live entities out on the network that can be easily changed without changing the clients—there is no need to push a software update to the users.

APIs are not much different in this respect. Assuming your API remains backward compatible, an API program can introduce new features and change the implementation of existing features without "breaking" the clients. As long as you uphold the contract between your API and the developers who use it, the API can change on a "web schedule" rather than on an "enterprise IT schedule." The result is a better, more responsive API program.

In fact, changes to both APIs and websites can be driven by analytics on behavior of the applications and end users. In both cases, a good design and product management team constantly checks the analytics to see what parts of the site or API are succeeding and which are failing. The result should feed into regular development sprints, which over time build a much more robust API or website.

Who Uses an API?

We call the company or organization that offers an API the *API provider*. This book is largely aimed at API providers (or those who are thinking about offering an API).

We decided to call the people who use your API to create applications *developers*. It's true that many types of people may be interested in your API, including business owners, marketing gurus, executives, and others, but the people who will eventually create the applications are developers. Developers are the primary audience for your API.

We decided to call the people who use the applications that developers create *end users*. They are the secondary audience for your API and often the audience driving many of your API decisions. Depending on the content made available via your API, you may have particular concerns to address, such as copyright, legal use, and so on, that relate to this secondary audience.

Types of APIs

We see two types of APIs: private and public. No matter what you may hear in the media, private APIs are the more prevalent variety. You know about the Facebooks and Twitters of the world and their use of APIs. What you probably don't know is that those same companies are likely making much more extensive use of their own APIs to drive their websites, mobile apps, and other customer-facing products. In our experience, visible public APIs like these are just the tip of the iceberg. Like the large underwater mass of an iceberg, most APIs are private and imperceptible, internal to companies, used by staff and by partners with contractual agreements. This use of APIs is what is really driving the API revolution. Do not limit your thinking about the ways APIs can be used to public examples like the App Store. Partner and internal use of APIs is often more valuable.

Much of the discussion of APIs assumes that they must be open to the public to be of value. This is not the case. We believe that private APIs are having a transformational impact on most companies, in many cases much more so than public APIs.

The *New York Times* API started as a private API and is transforming their business. "The NYT API grew out of a need to make our own internal content management system more accessible so that we could get the most from our content," said Derek Willis, Newsroom Developer at the *Times*. "The API offered a way to give more people access to create more interesting pieces. We are the biggest users of our own API, and that's not by accident. The API helps our business in other ways: creating brand awareness and helping us attract talent. But fundamentally, it helps us do our own jobs better."

To further frame this discussion, let's clarify what we mean by public and private. Public means that the API is available to almost anyone with little or no contractual arrangement (beyond agreement to the terms of use) with the API provider. Private APIs are used in a variety of ways, whether to support internal API efforts or a partner's use of the API. API providers also offer private APIs to large customers with appropriate legal contracts. Private and public really refers to the formality of the business arrangement. It doesn't refer to the API content nor does it refer to the applications developed using the API.

Finally, public and private APIs are, in the end, still APIs. Often a company will start with a private API and eventually open some or all of it for public access, possibly with additional restrictions. Other times, a company will launch a public API, then realize how important it is for internal development and in the end it is private use, not public use, that provides the real business benefit.

AccuWeather, for example, is well known for providing weather data to the general public, which would lead most to believe that their APIs are public. But remember: the private/public distinction refers to the arrangement with partners, not to the availability

In brief, tech blogger Robert Scoble summed up where we are now by defining three eras:

- Web 1994 was the "get me a domain and a page" era
- Web 2000 was the "make my pages interactive and put people on it" era
- Web 2010 is the "get rid of pages and glue APIs and people together" era

We believe that this profound shift will continue and that it's important for you to know more about it. Chapter 2 describes the impetus behind APIs as a business strategy.