A Practical Approach to API Design

From Principles to Practice

D. Keith Casey Jr & James Higginbotham
A Practical Approach to API Design
From Principles to Practice

D. Keith Casey Jr and James Higginbotham

This book is for sale at http://leanpub.com/restful-api-design

This version was published on 2016-02-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2013 - 2016 D. Keith Casey Jr and James Higginbotham
Tweet This Book!

Please help D. Keith Casey Jr and James Higginbotham by spreading the word about this book on Twitter!

The suggested hashtag for this book is #apidesign.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#apidesign
# Contents

Welcome ................................................................. 1

Changelog ............................................................. 2

Chapter 1 - APIs: An Introduction ................................. 3
  What is an API .................................................. 3
  Why you should have an API .................................. 4

Chapter 2 - Principles of Pragmatic API Design ............... 9
  API or SOA? ...................................................... 9
  API and SOA: Extend and Accelerate ....................... 11
  REST vs. SOAP ................................................ 11
  APIs for Existing Applications ............................... 13

Chapter 3 - Beyond CRUD .......................................... 15
  Building APIs for Business Workflows .................... 15
  Affordances ..................................................... 17
  Why you should trust others’ APIs ....................... 19

Chapter 4 - The API Modeling Process ........................... 20
  Summary of the Steps ........................................ 20
  Step 1: Identify the Participants ............................ 21
  Step 2: Identify the Activities .............................. 22
  Step 3: Break Each Activity Into Steps .................... 23
  Step 4: Create and Group API Methods ..................... 24
  Step 5: Validate the API By Testing ....................... 25
  Common Questions About API Modeling .................... 26
  Next Step: Design ............................................. 27

Chapter 5 - The Principles of HTTP and REST .................. 28
  HTTP Core Concepts ......................................... 28
  Web APIs and the Richardson Maturity Model ............. 32
  Back to Basics: The 6 REST Constraints ................... 35
  The Next Steps ................................................. 40
## CONTENTS

### Chapter 6 - From Modeling to RESTful Design
- Building Your Resource Taxonomy .............................................. 42
- Defining Resource Lifecycles ................................................... 42
- Mapping Response Codes For Success and Failure ..................... 45
- Expanding Resources Through Hypermedia Linking .................. 46
- Common Questions About API Design .................................... 50
- Next steps ........................................................................ 53

### Chapter 7 - Data Architectures
- Hypermedia Specifications ....................................................... 55
- Selecting a Data Storage Strategy ............................................. 58

### Chapter 8 - Documenting Your API
- The Hidden Value of API Documentation .................................. 62
- API Definition Formats .......................................................... 62
- Beyond API Reference Documentation .................................... 64

### Chapter 9 - Prototyping Your API
- Static Prototyping .................................................................. 66
- Building a Working Prototype ................................................ 67

### Chapter 10 - API Product Management
- The API-First Organization ...................................................... 69
- Front-End, Back-End, and Full Stack Developers ...................... 69
- Partitioning API Teams ......................................................... 71
- Accelerating API Integration .................................................. 71

### Closing.. for now .............................................................. 74
- Want to stay informed about everything API-related? ............... 75
Welcome

There are only two hard things in Computer Science: cache invalidation and naming things.

– Phil Karlton\(^1\)

Fundamentally, API design is hard. It combines two of the most challenging things in computer science and asks you to accomplish them in a distributed environment at scale. You might as well try it with your eyes closed, too.

In this book, we’ll discuss the What, Why, When, and How behind RESTful APIs. We’ll begin the story by describing what an API is. Then we’ll discuss business and technical scenarios that might make an API make sense for your organization. Then we’ll talk about how the core concepts fit together and the design principles that emerge. Throughout, we’ll share practical real-world examples of all of the above. While many are success stories which can be emulated, many will be cautionary tales of strategies and implementations to avoid. Finally, while we’ll touch on and explore the various religious aspects of API design, the goal of this book is to remain relatively neutral and give the different sides equal time.

While this is a book, we hope it also becomes a conversation. When you find something interesting, let us know. If you think we’re wrong, say so. If you think we’re right, tell your friends. If you want us to teach your team more and deeper concepts, please let us know.

D. Keith Casey, Jr. keith@caseysoftware.com
James Higginbotham james@launchany.com
February 2014

\(^1\)http://martinfowler.com/bliki/TwoHardThings.html
# Changelog

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 Feb</td>
<td>Initial publication</td>
</tr>
<tr>
<td>15 Mar</td>
<td>Fixing typos, some formatting tweaks, added the Closing</td>
</tr>
<tr>
<td>16 Apr</td>
<td>Fixing typos, some formatting tweaks</td>
</tr>
<tr>
<td>20 Apr</td>
<td>Added a section to Chapter 3 on “Why you should trust others’ APIs”</td>
</tr>
<tr>
<td>18 May</td>
<td>Updated the cover image</td>
</tr>
<tr>
<td>28 May</td>
<td>Merged Chapters 5 and 6</td>
</tr>
<tr>
<td>06 Aug</td>
<td>Finished Chapters 5 and 6, published a new edition</td>
</tr>
<tr>
<td>25 Feb</td>
<td>Finished Chapters 7-10, published a new edition</td>
</tr>
</tbody>
</table>
Chapter 1 - APIs: An Introduction

What is an API

If you read the tech press, everyone knows they need an API but most aren’t really sure what it is. They treat it as another checkbox like “Web 2.0” was a few years ago or a mobile app was most recently. In fact, there’s an entire “API-first” movement in development circles that most people don’t understand or even realize why. There are a variety of reasons for this situation with most reasons boiling down to a simple misunderstanding on what an API is and its purpose. Let’s get the obligatory Wikipedia definition out of the way:

An application programming interface (API) specifies how some software components should interact with each other.

– Obligatory Wikipedia Definition²

An API doesn’t have to be RESTful. It doesn’t have to use SOAP. It doesn’t have to use JSON or XML or OAuth or be built in Node or Rails or Python. It doesn’t have to have pretty URLs or even necessarily use URLs at all. It doesn’t have to make use of hypermedia concepts. The only requirement for an API to be an API is that it allows one program or software component to interact with another in a repeatable way.

It’s almost disappointing, isn’t it?

All of that said, an API can use all of those things listed above and definitely should use a number of them. When we speak about APIs in reference to web technology, it usually means a way of communicating over HTTP in a predictable and repeatable manner using payloads of XML or JSON to change the state of data within the system. More advanced systems use hypermedia and OAuth while less advanced or legacy APIs may use hard coded URLs and basic username and password-based authentication.

If that paragraph means nothing to you, don’t worry about it. We’ll get into it later. Many of the terms above are powerful concepts that help you build APIs in clean and predictable ways while also making it easy for your customers and users to adopt them. Because isn’t that what matters?

There are two types of companies that have APIs. First are the type of companies that have APIs. First are the type of companies that have APIs. First are the type of companies that have APIs. In this category, we find companies like SendGrid, Twilio, FullContact, and a host of others. In their case, the API is their business so it’s absolutely necessary that their APIs are easy to use and quick to get started. The other category is businesses who have an API as one of their products. In this group we find Walgreens, Best Buy, FedEx, and many other traditional offline companies looking to expand their reach and presence online. It’s worthwhile to remember that in many of these cases, they derive no direct revenue from API usage, which affect their strategy and implementation in a variety of ways.

²http://en.wikipedia.org/wiki/Application_programming_interface
At the end of the day, you must understand which category your organization will occupy. If usage of your API is directly connected to revenue, the goals are plain and simple: get people started quickly, make sure they keep using it, and give them reason to embed it throughout their organization. If an API is a checkbox that doesn’t serve a business or technical case, it is doomed to failure. Luckily there are numerous business and technical cases your API could satisfy, regardless of your industry or position in the market.

**Why you should have an API**

If you believe the tech press, you should have an API because...everyone should have an API. While this is great for the API management companies, it doesn’t really apply to everyone. If an API doesn’t provide tangible value to at least one major constituency within your organization, it is doomed to failure. Here are seven primary reasons on why an organization might need an API.

**Business Cases**

**Mobile Strategy**

The first business reason for an API is a mobile or mobile-first strategy. In most organizations, there are neither the time nor the resources to build tools for every single device. An API can ensure that additional - either less important or even unexpected - devices are addressed. In addition, it may parallelize the effort to create a presence much faster or differently than originally planned.

A real-world example is Netflix. When Netflix originally started mailing DVDs in 1997, its competition was Blockbuster and the cable providers. Over the next few years, Netflix quietly consumed the DVD rental market while they prepared to launch their streaming service in **February 2007**. Seventeen months later in **October 2008**, they launched the official Netflix API with a simple goal and a strategy to get there by partnering with developers:

*Millions of people love movies via Netflix, making this API an opportunity for all kinds of developers to add well-known value to any other application.*

*The company says the API will allow access to data for 100,000 movie and TV episode titles on DVD as well as Netflix account access on a user’s behalf.*

As a result, there were Netflix applications for most common devices within weeks or months. These initial applications provided customers with the ability to view and manipulate their queue

---


whether they were online or offline. More importantly, it began to make Netflix a tiny - yet constant - part of everyday entertainment consumption. By the time Blockbuster entered the streaming market and later when they launched an API, Netflix was already the defacto standard. By that point, if a device had a full color screen, no matter its manufacturer, size, or profile, there was a Netflix application available that could enable watching a movie in just minutes.

**Drive Usage**

The next business reason for an API is to drive usage of a platform. Notice that this isn’t necessarily paid usage but usage in general. Why is this scenario useful? By focusing on fast and simple usage of a platform, you can drive engagement and grow the user base even more.

Twitter is a prime example. Their API was probably one of the most commonly cited for simplicity and ease of use. It was so easy that many coding frameworks changed their sample “getting started” project from “hello world” to “hello twitter.” In a matter of months, there were literally hundreds of Twitter apps available on every device, ranging from desktops to browsers to more unique devices.

Twitter was not the first instant messaging or chat app and it has had to deal with competitors since launch. But by being ubiquitous on every device, it allowed people to step away from their computers and engage with the platform how and when they wanted. As a result, people used the platform more often than the alternatives. As people used the platform more, more people had to stay connected to it more often, further driving both passive (viewer) and active (tweeter) usage. This became evident at [South By Southwest 2007](http://mashable.com/2011/03/05/sxsw-launches/) when a mobile-centric, desktop-disconnected audience found it as the primary means to trade information, keep in touch, and find out what others were doing.

Ubiquity may not have been their original goal for the API, but it created a virtuous cycle of usage driving viewing and viewing driving usage.

**Defensive**

The third business reason for an API is purely defensive. This is somewhat parsing words from the previous two scenarios but is distinct in a number of ways: when you’re an established player in the market and competition disrupts your business, it can be a challenge to use your positioning, information, and infrastructure to make your existing business lines more stable. An API strategy may be able to launch you into new business lines and growth that was previously unrecognized.

The big box retailers have suffered the brunt of this sort of disruption. They have massive stores with massive inventories where they suffer from the “showrooming” problem. Potential customers come in, browse the merchandise, and once they find the item, they use a site like Amazon to find a better price and place the order. The customer walks out without the product but saved money and the retailer missed a potential sale.
Best Buy is working to turn that tide. By opening their API⁶ and making massive amounts of data open to third parties, they have been able to build a community that is willing, and even excited, to explore the data and ways that it can be used. Some of the apps range from simple inventory explorers, but others include republishing the deals and discounts they are offering in store. Yes, people are sharing ads on purpose. To date, it has had a significant impact on their bottom line with a 15% increase in online sales in Q3 2013⁷. Note that this is before the majority of Christmas shopping days, such as Black Friday.

Partner Connectivity

The final business reason is to improve connectivity with partners and suppliers. In the past, many organizations exchanged spreadsheets via email to update systems such as inventory management. If they were more advanced, the spreadsheets were uploaded via FTP and parsed automatically. Either way, these methods were error-prone and often fragile when changes would occur within either system. When importing a spreadsheet, a missing or new field, or even a misplaced comma, could cost hours of effort and delay expensive processes. Fundamentally, it was broken.

Alternatively, using APIs can provide a repeatable and consistent way to exchange information. The FedEx API demonstrates this from two different angles. First, by providing tight integration, an organization can simplify their own logistics by building on those within FedEx. On the other side, the organization’s customers gain a great deal of insight into shipping, which was an opaque process just a few years ago.

Technical Cases

Abstraction of Complexity

Powerful and important systems are almost always complex to use and integrate, often out of necessity. In the earliest days of the web, it took knowledge of the C programming language and detailed understanding of SMTP in order to send an email. As these systems have become more important and widespread, they have become easier and less complex to use and integrate into our systems. This ease and simplicity comes from APIs.

⁶https://bbyopen.com/developer
⁷https://www.internetretailer.com/2013/11/19/best-buys-online-sales-increase-151-q3
At this stage, the code required to send an email is quite simple. Unfortunately, this simplicity doesn’t include clickthrough tracking necessary for many emails. It also doesn’t include configuring email servers that are resistant to both sending and receiving spam, delivery reports, open reports, and a variety of other aspects that are useful to you and your organization. Every single one of those features is additional code to write and maintain in the future.

On the other hand, SendGrid has a simple API for sending email. It only takes minutes to integrate into a system. The fundamental difference is that it does all of the above features - clickthrough tracking, spam resistance, delivery & open reports, etc - automatically. You don’t have to do anything extra, just use the API. The API takes what is otherwise a complex system and allows you to make use of massive functionality in mere minutes with just a few lines of code.

**Simplifying Interfaces**

Despite numerous technical standards and long-established players, some systems are still terribly complex to interact with and use in general. One of the prime examples is your email inbox. Despite only two standards - POP3 and IMAP - there are so many implementations and variations of those implementations that what should be a handful of lines of simple code ends up being a deceptively complex implementation handling dozens of edge cases and unique scenarios. Building yet another email interface is a practice in futility likely to cause unforeseen schedule delays and unnecessary complexity. It’s simply not worth the time or energy.

In this scenario, an API can wrap all these variations and oddball cases and instead present a single unified interface for developers to use. The Context.io API does exactly this. To interact with an inbox - whether Microsoft Exchange, Gmail, or something else entirely - you initialize the inbox with your email credentials and then it transparently handles all the detail, complexity, odd scenarios, and other situations that occur. What could easily take days of effort is simplified to minutes. More importantly, it doesn’t matter if the inbox is hosted in Microsoft Exchange, Gmail, or anything else, all are used exactly the same way with the exact same commands. The most powerful aspect is that the developer doesn’t even have to know which email provider is being used. It’s completely transparent.

**Metered Usage**

Many services lend themselves to simple monthly or flat rate pricing. For example, sending email has a marginal cost that is effectively zero so sending the hundred and first email costs nothing
more than the hundredth. Alternatively, if we consider a resource such as CPU or processing time, the costs of running a server for two hours is effectively double the costs for one hour. In this case, a consumption-based or metered pricing model makes more sense. This is also known as “utility pricing” to represent that you are only billed for the portion of a resource used.

This is key for scenarios where massive amounts of processing time or power are required for short periods. A few years ago, the New York Times had just this situation⁸ where they had 130 years of articles, approximately 11 million in total, which needed conversion into PDF documents. To solve the problem, they turned to Amazon EC2 to increase their processing power. As a result, the Times was able to temporarily utilize 100 servers to generate all of the required PDFs in under 24 hours. At the end of the project, the Times released those servers and was not charged any further.

All of the above

Of course, none of these scenarios are mutually exclusive. An API can be used to simplify interfaces, hide complexity, and still create a system for metered usage.

The Twilio API is an ideal example of addressing all of these challenges. Interacting with the telephone network is hard. There are numerous standards implemented at any given layer and each carrier implements those standards just a bit differently. To add to the complexity, carriers in different countries implement different standards. And to top it all off, the same carrier in different countries can use different standards to accomplish the same task. Further, interacting with all of these systems and components can be quite expensive. By combining all of these scenarios together, we can have a single API that abstracts away complexity and simplifies interfaces while limiting overall costs.

While the business cases for an API can be hard to quantify and therefore even harder to measure, the technical cases for an API are relatively simple and straightforward. Most developers understand building systems to make use of complex systems is a painful experience that they’d rather avoid. More importantly, they understand that replicating these systems in house often requires a huge amount of time and effort and is error prone. At the end of the day, building and using these APIs can save time, money, effort, and untold amounts of stress for all involved.

Chapter 2 - Principles of Pragmatic API Design

There have been a variety of technologies promising rapid development, code reuse, and better integration. Most recently, enterprises have adopted a Service Oriented Architecture (SOA) in an effort to build reusable web services that can be combined to solve a variety of problems. While many enterprises are reaping the rewards of this shift, SOA only solves some of the problems for today’s CIOs and CTOs.

Meanwhile, companies such as Google, Amazon, numerous startups, and even individual developers are releasing APIs that enable just about anyone to create, improve, and collaborate with minimal effort.

How does today’s technology leader determine how to handle this new API-focused world? It first requires an understanding of the principles of API design.

**API or SOA?**

There is much confusion on APIs vs. SOA. Google search indicates there are over 1.9 million search results trying to address this specific issue. Many see them as the same thing, while others say you must choose between APIs and SOA. So, what is the right answer and what does it mean for your business?

To better understand the differences and how it all fits together, a quick review is needed.

**What is SOA?**

Service Oriented Architecture (SOA) is a set of architectural design principles common to large-scale software products and enterprises. The goal is to decompose an application into independent services that are decoupled, discoverable, and reusable. Often, a service oriented architecture will be composed of a variety of systems, a variety of services, and connected via a message bus. This approach allows each system to be integrated without the internal services being aware of the other systems. It also allows for the combining of various services to compose completely new solutions.

The modularization of SOA is similar to the best practices of decomposition and loose-coupling that are found when building standalone applications. However, SOA makes some additional assumptions:

1. Services are on a network, rather than within a single process
2. Services communicate through messages, rather than procedure calls
3. Services can be discovered via a directory service
4. Services are defined by a contract, ensuring that clients know exactly how to send and receive properly formatted messages
5. Services must be governed to apply best practices, adhere to architectural principles, and conform to necessary government regulations

Traditionally, using SOA required one or more architects that defined the systems, service contracts, and messaging. In addition, one or more teams are required to implement each service. Due to the difficulty of designing, implementing, and integrating services in this manner, many software companies and enterprises were forced to move slower than with traditional software architectures. As a result, SOA had a higher cost and often required years to actualize the ROI. Only recently have these companies started to realize the benefits of their investment.

**What is an API?**

An API, or Application Programming Interface, specifies how components within software should interact with one another. Prior to web services, APIs were typically delivered for a specific programming language and offered as a library that would be used to integrate new functionality into an application.

SOA pushed APIs to be seen as network services (“web APIs”), rather than just programming language libraries. This created two roles with SOA: service providers and service consumers. Standards such as SOAP were published to provide rules, contract definitions, and structure around the underlying protocols. The result was that web service APIs became the glue that held SOA together between providers and consumers.

While SOA was solving the problem of building large-scale software and enterprise systems through web service APIs, many smaller software vendors found the approach to be too cumbersome and costly. Plus, with the growth of mobile devices that only offered limited network access and processor speed, a leaner approach to consuming services was required. Finally, APIs needed to move outside the firewall and SOA was inward-facing.

To meet these changing demands, software vendors started to take portions of the service oriented architecture and applying it in a more lean fashion. These next generation web APIs removed many of the SOA rules to make things faster to build and consumable both inside and outside the firewall. The result was a leaner set of guidelines for building APIs:

1. Focus on the human rather than on machines
2. Focus on collaboration over governance
3. Focus on solving integration with partners first, then handle internal integration needs as required

This leads to a focus on simplicity rather than service governance:

1. Remove complex networking protocols, such as SOAP, in favor of REST that is built directly on the HTTP standard
2. Remove the need for directory services, contracts, and a message bus to reduce complexity where possible
3. Remove the XML-only approach, looking to JSON for faster transmission and parsing on browser and mobile platforms
With many of the obstacles removed, API adoption grew. The result, according to a 2013 infographic from Layer 7⁹, is that 43.2% of organizations already have an API program in place, and 86.5% state that they will have an API program within five years.

The question is now moving from, “Should we build an API?” to “What APIs should we build first?”

API and SOA: Extend and Accelerate

The world of APIs is exploding, so where does this leave SOA? Like many trends, they often start at one extreme of complexity and bounce to the other end. Over the recent years, we have seen companies move from SOA to simply putting an API on their software. Moving forward, you will see the trend settle somewhere in between, with software being defined with an API-first approach while being built using a service oriented architecture.

While APIs can be built without a service oriented approach, there are a number of drivers pushing for another look at the principles of SOA:

1. The growth of single page applications (SPAs) within browsers that require APIs to access data and business logic
2. The growth of multi-device applications that work across browsers, mobile devices, and the emerging world of connected devices (called the Internet of Things or “IoT”)
3. The need for robust, bi-directional and asynchronous web service communication through webhooks and data streaming
4. The increased complexity and need for lightweight governance when taking an API-first approach
5. The desire for a more contract-driven approach to API consumption, for both more robust automation and tooling support

As a result, businesses should be looking at both APIs and SOA. APIs extend the reach of your software and data; SOA accelerates it through proven patterns.

REST vs. SOAP

SOAP is an industry standard that defines the protocol for web service definition and consumption. It has been around since the late 1990s and is currently on v1.2 of the specification. SOAP has been, and continues to be, popular within enterprises and therefore is a common choice with software that seeks to offer enterprise integration. The SOAP specifications offer strict data type checking and multiple protocols for both synchronous (RPC-style) and asynchronous (fire-and-forget) invocation.

SOAP has the following advantages:

1. Support of multiple protocols (HTTP, SMTP, and JMS) and requirement of a single payload format (XML)

⁹http://www.layer7tech.com/infographic/img/inf-2-download.jpg
2. Built-in semantics for asynchronous processing, rather than just the traditional RPC-based request-response
3. Formal contracts for the definition of services and contained messages, including data types
4. Support for state, allowing for complex workflow scenarios and cross-service transactions
5. Optional specifications for reliable messaging, transactions, coordination, and security beyond the base SOAP specification

Many software companies have found that implementing anything beyond the basic HTTP SOAP specification required investment in both third-party software and an increase in development expertise. It also increases the time to adoption, as SOAP API consumers must select a SOAP client library, create their own, or wait for the vendor to provide a library in their preferred programming language. This can severely limit the adoption rate of APIs.

The paper “Architectural Styles and the Design of Network-based Software Architectures”¹⁰ by Roy Thomas Fielding has changed all of this, introducing companies to a series of patterns that offers software companies a faster and easier way to build and integrate web service APIs. Instead of defining of set of standards that supported multiple protocols and payload formats, Fielding suggested the use of the HTTP specification itself as a starting point. Since there are a variety of HTTP verbs already (e.g. GET, POST, PUT, DELETE, PATCH), and response codes provide details on the success or failure of the call (e.g. 200 OK, 404 NOT FOUND, etc) most of the work was already done. Services can accept incoming payloads in a variety of formats, using HTTP request headers to indicate the type(s) of data formats desired including media such as images and videos. Rather than requiring a specification, Fielding encouraged the use of common patterns on top of these specifications. Calling it REST (“Representational State Transfer”), Fielding suggested that services should be stateless, just as HTTP is stateless, further simplifying the service implementation requirements.

REST has the following advantages:

1. Stateless operation, minimizing implementation details
2. Limited bandwidth requirements
3. Support for caching due to its stateless nature and HTTP caching standards
4. Flexibility regarding payload structures, including popular formats such as XML and JSON
5. Ease of use, often with only an HTTP client library rather than a custom library that supports the SOAP protocols

REST, however, does have several disadvantages. Since it is a style rather than a specification, API implementations may vary and therefore may require a custom client for each service. In addition, it is only designed for request-response support by default, requiring asynchronous support to be custom built and therefore varying between API vendors. However, the use of existing specifications and limited development overhead make REST the default decision for most web APIs.

So, is it SOAP or REST? Actually, it can and might be both. SOAP offers specific semantics that work better with enterprises that already utilize existing SOAP-based tools and expertise.

addition, it can be used to transport messages outside of HTTP, providing more flexibility regarding
the types of clients that will be consuming the API. However, offering all or a portion of your API
using REST provides easier adoption for common APIs that do not have these requirements. Also
keep in mind that many vendors have removed, or in the process or removing, their existing SOAP
APIs in favor of the newer and more accepted REST style.

REST is an architectural style that makes it fast to model, design, and implement web service
APIs. SOAP is a protocol standard that is designed for interoperability in specific circumstances that
otherwise isn’t provided for with REST. Our suggestion is to start with REST-based APIs and then
evaluate if there is a need for SOAP-based APIs. Over time, REST will likely be supplemented with
optional specifications or recommendations that will offer SOAP-like semantics when necessary.
Examples of this already happening include: the use of Swagger¹¹ or RAML¹² to define REST
API contracts, the introduction of Webhooks for asynchronous callbacks, and the growth of API
management platforms for policy and access management.

**APIs for Existing Applications**

Taking an API first approach means designing and developing an API to provide a consistent user
experience across a variety of platforms. By taking this approach, companies begin to look at the
higher-level user workflows while still working out the details of the user interface. The result is
an application that is prepared both for a great user experience (UX) through the user interface, but
also a great developer experience (DX) through the API.

While this is useful for new applications, how does an existing application take advantage of
building APIs to grow their business? Given that many existing applications were not built using a
service oriented architecture, adding an API can be difficult. However, there are three strategies for
adding an API to existing application, allowing you to take advantage of APIs for your business.

**Bolt-On Strategy**

The Bolt-On Strategy looks for opportunities to build APIs for reading, and optionally writing, data
into your application by directly adding API support into the application. Often, this data may be
in a more cleansed form, obtained either via a data warehouse, an ETL process, or directly from a
transactional data store.

The point of these data sources is that APIs should be designed to offer high-level data that is
both useful and actionable by API consumers. Often, these APIs are supplemented with additional
third-party data, making the insights more valuable.

This strategy is the most brute-force approach, but often the best way to get started offering
an API as it allows teams to capitalize on existing code and business logic contained within an
application. However, the downside of this approach is that the application can become fragile or
unwieldy for large systems. Often, many systemic problems within the application that were made
in the past begin to “leak” into the API and can create a poor developer experience. Therefore, use
cautions when utilizing this strategy.

¹¹https://helloreverb.com/developers/swagger/
¹²http://raml.org/
**Greenfield Strategy**

The Greenfield Strategy looks for new product offerings and begins to apply an API-first approach to an existing product line. When possible, these projects benefit from being built using newer technologies that accelerate the development process. This can reinvigorate an existing development team and is also useful as a recruiting tool to attract fresh talent that desires to work with the latest technologies.

When selecting new technologies for your project, it is best to identify those that will allow for easier integration into the existing product. Not only will this ease the transition into the new API, it will often improve time-to-market as existing investments can be reused. An example might be the use of JRuby and Sinatra for building out an API for a product currently built on Java-based technology, providing code reuse while retaining familiarity for operations and technical support teams.

**API Facade Strategy**

This API Facade Strategy is similar to taking an API first approach, as it requires that you design the API fully as if you were building it from scratch. Then, the API is implemented by calling into the existing application as necessary. Over time, the old portions of the application are removed and replaced with a more service-oriented architecture until the legacy code is “strangled out” and no longer remains. This approach is often considered lowest risk for legacy systems, as the existing application is kept intact, but new APIs are built and deployed over time on top of it.

This strategy is also common for companies such as HomeAway\(^\text{13}\), that have a merger and acquisition (M&A) strategy. These companies must integrate a variety of acquired systems into a single platform. An API facade can be designed and wrapped around each acquired technology stack, providing a single way of integrating a variety of acquired technologies into a cohesive platform.

\(^\text{13}\)http://www.homeaway.com/
Chapter 3 - Beyond CRUD

Building APIs for Business Workflows

Most APIs are built around the concept of simple CRUD or Create, Retrieve, Update and Delete. They are built with the assumption that most people will need to create some sort of record, read it back later, update it at some point, and delete it when they’re done. In simplest terms, this is all many APIs will need to accomplish many business workflows. But once you think more about the individual actions within the business, you’ll quickly realize that while most operations look like CRUD, they are a little more intricate and there are numerous steps and processes hidden just under the surface.

Think about ordering a cup of coffee: you place (create) your order with the barrista, wait a few minutes, and then get (retrieve) a hot cup of coffee on the other end. This is obviously a CRUD operation until we consider what is happening behind the scenes. Looking a little deeper, we find a hint of complexity:

- You create an order with the barrista;
- the barrista passes the order to the person running the machine;
- the person acknowledges the order and adds it to their queue;
- as those orders are ready, they’re announced and delivered to the counter;
- customers retrieve their beverages.

Even this process hides the fact that throughout this entire process, a financial transaction occurred from each order; the barristas kept a close eye on their supplies, equipment, and ingredients; a few people changed their order; and someone canceled an order. What began as a simple three-step CRUD process (order, wait, receive) has spiraled into a complex workflow dependent on various ingredients, special equipment, numerous people, and a variety of tasks of various durations. Imagine what it takes to receive a pizza delivery!

Once we look at our business workflows, we’ll realize that while CRUD often works, it is just the tip of the iceberg. We need to be aware of the inherent complexity that goes along with almost any business process. This complexity can include numerous people, multiple business groups, online and offline processes, and even entirely separate companies. If we ignore the complexity, we run the risk of building systems and processes that do not actually meet business needs and are regarded as failures.

Toward that goal, we need to shift our thinking from CRUD to thinking about goals and the sequence of actions they require. For example, we would never describe “getting a cup of coffee” as a CRUD operation. Instead, we need to think in terms of goals: getting a cup of coffee. Once we dig a little deeper, we can quickly recognize the actions that go into it. Many are CRUD but the vast majority are related to either a) waiting for something or b) modifying one of our somethings based on other events occurring.
Thinking in terms of actions also gives us two other benefits:

First, we can speak English. Too often we get stuck in the technical terminology describing what we are doing instead of describing the why. Once we can speak in terms of why then anyone on the team - from the least technical to the most - can understand exactly what is happening and why.

Second, we can make the processes more flexible in general. In the real world, whether you get coffee at Starbucks or your local coffee shop, the steps are fundamentally the same even if they’re accomplished a little differently. By separating the why from the how we can see the similarities and figure out which pieces are interchangeable.

This is usually accomplished by limiting jargon used to describe the processes. For example, a scenario might be described as this:

“A user will go to the Create Account Page and use the Create Account Service to create an account. Later they may choose to edit their profile via the Account Manager to change their email address.”

While this is technically accurate, we’re describing the how in terms of the components and architecture we have now. It assumes that whomever is reading this has a huge amount of information and understanding about the system. If we change or modify our architecture at a later time, we have to go back and edit this scenario so it still makes sense. More likely, as time passes, no one will maintain these scenarios and we’ll have yet another place for outdated information and incomplete documentation. Let’s modify this scenario to remove the how:

“A user will create an account. Later they may choose to change their email address.”

Notice the two statements both explain the same scenario but by removing the how, our second statement is more succinct. It also doesn’t require the reader to know much about our system such as how it is built or which individual screens or services might exist. In the first scenario, it was strongly implied that each user would create their own account. The second scenario allows this but may also allow for an administrator to create an account for someone else. Now we need to decide if we want to allow that. So our statement could become:

“A user will create their own account. Later that user may choose to change their email address.”

and possibly:

“An administrator can create an account for a new user.”

And independently of both cases, we have another statement:

“A user may choose to change their email address.”
Although all of these statements come from the original requirement, we couldn’t see them clearly through all the noise and jargon of the “how.” Once we move outside the realm of how, we may notice ways to simplify, clarify, or even eliminate some of the steps involved. At the end of the day, this subtle difference can let us build the system easier, faster, and with more confidence that we’re actually satisfying real business goals. We may also find additional requirements and details hidden within the original scenario which were assumed by the customer and overlooked by the developers. Simplicity and clarity can literally save the project.

If you want to have a fighting chance to get users to use your API, the design must extend past the normal CRUD operations and instead express actual business workflows of tasks your users want to accomplish.

**Affordances**

An afford is a property of an object, or an environment, which allows an individual to perform an action. For example, a knob affords twisting, and perhaps pushing, while a cord affords pulling.


When we talk about affordances with respect to APIs, there are three aspects that should align: first, there is what the API is designed to do; second, there is what the API makes easy to do; finally, there is what people actually use the API to accomplish. Let’s take each of these in turn.

The classic example is a coffee cup. A coffee cup has a handle designed to make it simple to hold, even when the contents are hot. As a result, most people hold it by the handle. In this case, the design, easy to use path, and what people actually do are all aligned. The handle may be used for other purposes - such as hanging the cup - but its primary function is obvious and easy to understand. While we can all think of numerous examples where all three aspects align, there are many more combinations where they don’t align and are even in direct opposition to one another.

Alternatively, when email was designed, the Internet was a completely different beast than what we know now. The Internet was designed specifically for communicating scientific data between universities and the military. More importantly, everyone who had access was known by their actual name, therefore security and verifying identity were unnecessary and not considered in the design stages. Instead connectivity and resilience in the case of failure or nuclear attack were the most important. Therefore, the design and all the supporting systems were optimized for this scenario. Fast forward a couple decades to the widespread public use and connectivity of the Internet and - due to those early design decisions - malicious users can connect inappropriate services that create problems such as spam, fraud, and denial of service attacks. The original affordance was completely successful but it had significant longterm drawbacks: what was a feature in one form became a weakness in yet another.

As you design an API, you must remember that what you design it to do and what tasks you make easy should be the same tasks as what your users actually want to accomplish. Unfortunately,
that’s much easier said than done. Even in the best scenarios, it’s incredibly hard to do when you’re building the first version of your API. No matter how much research, effort, or planning you spend on your API, the first version will be incomplete at best and wrong at worst. This is not a reflection of your organization or team. Instead it’s a reflection of what you don’t know about your current and potential customers, their overall goals, and how they go about those goals. Most likely, you’ve talked to customers to find out what they do but it’s very hard to ask what they might do once they have new, unexpected tools available.

“If I had asked people what they wanted, they would have said faster horses.”

Henry Ford, supposedly

In the API world, a good example is Twilio’s conference call system. Putting people on hold is a common scenario for almost any business. Originally there was only one way to do this with Twilio: you would place the caller into a conference call muted. Then you would place a second call into the conference to play your desired hold music. In concept, it was a relatively simple solution to a common problem. In implementation, it required passing calls back and forth, keeping track of call states, who was connected, and numerous other aspects. Getting simple information like average hold time or who was next in the queue required dozens of API requests and more business logic than most were willing to deal with. It simply wasn’t designed for this scenario.

Seeing this, Twilio had a simple choice: They could either leave the status quo and force people to do these API gymnastics or they could build a better affordance to make this simpler. In this case, they built a call queuing mechanism that allowed you to add people to the queue with a single request. You could remove the next caller and connect them to an agent with a single request. And finally, you could collect statistics with a single request.

In terms of technical details, calculating statistics like average wait time with the Conference-based implementation requires 3 requests to set up plus one request per caller in the queue. In mathematical terms, this is an O(n) operation. Alternatively, collecting the same information from a Queue-based implementation takes requires exactly one request regardless of how many people are on hold. Which would you want to use?

*https://github.com/caseysoftware/twilio-call-queue-howto

There is one final aspect of affordances that must be considered. While we would like to ensure that our APIs can be used to accomplish anything our customers might decide is important, our API can’t be everything to everyone. Every time we add an additional affordance, feature, piece of functionality, etc, we must support it for the lifetime of the API. If we attempt to build the Swiss Army Knife of APIs, we may create unnecessary maintenance and support issues that drain the organization of time and resources to build the next affordance. There’s a complex balancing act involved between affordances and maintenance which we will begin to address next.
Why you should trust others’ APIs

One of the most common questions that comes up at this stage is:

*Why should I trust someone else’s API?*

It’s a fair question and should be considered in relation to everything we’ve addressed so far. But the real question is more visceral and gets to the core of the matter:

*How do I make sure this company doesn’t turn off their API and leave me worse off than I was before?*

An API should first and foremost solve a business or technical issue. If you can’t see how it solves an actual problem for the provider, you should be at least hesitant and at worst skeptical of the API. In this realm, there are two primary types of companies which we’ll refer to as “API-based” and “API-also” companies.

The first group or “API-based” companies are companies like Twilio¹⁵, SendGrid¹⁶, Stripe¹⁷, and many, many others. In all of these cases, these companies are not just API providers but they’re entirely API companies. They only make money when people successfully use the API. Therefore, it is in their best interest to make the experience as smooth, fast, and as easy as possible. Further, since the API is their only product, they could not turn it off to focus on other products.

The second group or “API-also” companies include companies like Walgreens¹⁸, Best Buy¹⁹, Twitter²⁰ and many others. This second group is more concerning. In most cases, their API is in addition to their normal product offerings. Luckily, in some cases - like Walgreens and Best Buy - they derive revenue directly from API usage and activity. As long as those numbers continue to grow and open new business opportunities, the API should have a long, stable lifecycle.

Alternatively, there are companies like Twitter or Facebook who tie the vast majority of their revenue to services and offerings that make up a tiny percentage of their API users. This is the group of API providers are going to be the most problematic. They’re more likely to change the API - breaking existing clients - deprecate important pieces or even turn off the API entirely to offer exclusive features and functionality to partners. Before you consider integrating with one of these API providers, you should carefully consider your options and your approach. Later in this book we’ll discuss strategies for isolating these changes and protecting your organization.

---

¹⁵https://www.twilio.com/
¹⁶http://sendgrid.com/
¹⁷https://stripe.com/
¹⁸https://developer.walgreens.com/apis
¹⁹https://bbyopen.com/developer
²⁰https://dev.twitter.com/
Chapter 4 - The API Modeling Process

A great developer experience (“DX”) of your API is essential and ensures that developers can quickly and confidently consume your API. It also improves your API traction by moving developers from being integrators to becoming experts that deliver real value to their customers and their business. However, building a great developer experience for your API requires a careful approach to design.

Just as a beautiful web design begins from a wireframe, a great API design begins with modeling. The goal of API modeling is to fully understand and validate the needs of your developers and the end users. But unlike a wireframe, which focuses strictly on the end user interaction, API modeling focuses on both developer and end user goals. Sometimes these goals are aligned, but often they are not. API modeling ensures that both sets of goals are met, prior to designing and developing your API.

While API modeling is a comprehensive subject, this chapter will introduce you to some basic techniques for effectively modeling your API, ensuring that your API design meets the goals of both end users and developers.

Summary of the Steps

API modeling consists of 5 activities that help identify the requirements of your API design:

1. Identify the participants, or actors, that will interact with your API
2. Identify the activities that participants wish to achieve
3. Separate the activities into steps that the participants will perform
4. Create a list of API methods from the steps, grouped into common resource groups
5. Validate the API by using scenarios to test the completeness of the API

The goal of each step is to explore the requirements of the API from a variety of different perspectives: those involved (the participants), what they want to do (the activities), and how they will complete an activity (the steps). The modeling process will be iterative, so it should be expected that each modeling activity may require that you revisit the previous step as previously hidden details are exposed.
After you have completed these API modeling activities, you will be ready to move into API design. This is where you will map your API model into a well-designed API that handles both success and errors using pragmatic REST principles.

Remember: While this process might seem a little daunting or excessive, the goal is to build an API that developers will love. So, have fun with it!

**Step 1: Identify the Participants**

The first step in API modeling is to create a list of participants or actors that will interact with your API. This may be one or more of the following:

1. Internal developers
2. External developers
3. System administrators (i.e. your company’s administrators)
4. Account administrators (i.e. your customer’s administrators)
5. Users of the system (users, managers, moderators, etc)
6. Applications
7. Devices

Beginning with activities can lead to missed or incorrect APIs that fail to accommodate all types of participants. Listing the participants first improves the coverage of activities and helps to provide a more holistic view of how the API will be used.

For each participant, you will capture: who they are, if they are internal or external to your company, and a short description of who they are. For example:
If you are uncertain where to start, begin with a list of specific people or roles. Then work to generalize them into a common list of participants that best describe the types of people that will be interacting with the API (directly or indirectly). Also, keep in mind that internal and external systems that will integrate with your API may be considered participants as well. This is often the case with notifications and automated workflows.

With your list of participants in hand, we move to the next step: identifying the activities.

**Step 2: Identify the Activities**

An activity is work that produces a desired outcome and that is accomplished by one or more participants. Examples include: Place Order, Lookup Order Status, View Order History, Fulfill Order, and Resolve Order Problem. Notice that these activities likely include one or more steps required to accomplish the activity. While we will go into the “how” level of granularity soon, start by looking at outcomes or goals rather the steps required to achieve the activity.

Your API may focus on only one activity; however, most APIs focus on more than one. A single activity may require one or more participants, depending on the type of collaboration involved (e.g. internal-to-internal, internal-to-external, external-to-internal, external-to-external).

For each activity, you will capture: the name, the collaboration type, the participant(s) involved, and a short description. For example:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Participant(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Order</td>
<td>Customer</td>
<td>A customer places and pays for an order</td>
</tr>
<tr>
<td>Lookup Order Status</td>
<td>Customer, Manager,</td>
<td>Lookup an order status</td>
</tr>
<tr>
<td></td>
<td>Customer Support</td>
<td></td>
</tr>
<tr>
<td>Cancel Order</td>
<td>Manager, Customer Support</td>
<td>Cancel an order and refund payment</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

You may approach activity discovery in a variety of ways: UI-driven, participant-driven, or process-driven. Using more than one approach is recommended, as it will often yield previously hidden or forgotten activities as you view the problem from different perspectives.

You may also discover new participants during this exercise. If so, simply go back and add them to the list of participants. Then, revisit the new participant to see if any other activities are missing.
and need to be captured, or if they participant in other activities that you previously identified.

### Step 3: Break Each Activity Into Steps

Activities are composed of steps, with each step being accomplished by a single participant. Activity steps may be conducted by more than one participant, but a single step should only be executed by a single participant at a time.

Decomposing an activity into its individual steps requires a deeper understanding of how your API will solve real-world problems. This insight is generally left to a subject matter expert (“SME”). However, if a SME is unavailable, techniques such as interviewing other SMEs and customers can yield similar results. Just be sure to allow more time for this kind of research.

For each activity step you identify, you will capture: the activity, the step name, the participant(s) involved, and a short description. For example:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Step</th>
<th>Participant(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Order</td>
<td>List Items</td>
<td>Customer</td>
<td>List/search for store items</td>
</tr>
<tr>
<td>Place Order</td>
<td>Add Item To Cart</td>
<td>Customer</td>
<td>Add an item to the customer’s cart</td>
</tr>
<tr>
<td>Place Order</td>
<td>Remove Item From Cart</td>
<td>Customer</td>
<td>Remove an item from the customer’s cart</td>
</tr>
<tr>
<td>Place Order</td>
<td>Clear Cart</td>
<td>Customer</td>
<td>Remove all items from the customer’s cart</td>
</tr>
<tr>
<td>Place Order</td>
<td>View Cart</td>
<td>Customer</td>
<td>View the current cart and total</td>
</tr>
<tr>
<td>Place Order</td>
<td>Checkout</td>
<td>Customer</td>
<td>Create an order from the contents of the cart</td>
</tr>
<tr>
<td>Lookup Order Status</td>
<td>List Orders For Customer</td>
<td>Customer, Manager, Customer Support</td>
<td>List a customer’s orders from most recent</td>
</tr>
<tr>
<td>Lookup Order Status</td>
<td>Obtain order status</td>
<td>Customer, Manager, Customer Support</td>
<td>View the status of an order</td>
</tr>
<tr>
<td>Lookup Order Status</td>
<td>Locate Customer</td>
<td>Manager, Customer Support</td>
<td>Find a customer by name, ID, phone</td>
</tr>
</tbody>
</table>

As you go through each activity, you may notice that you are capturing steps that are duplicate
or similar across activities. This is an expected result and will help you to identify common API methods during the next modeling step. It may also help to review step names as you go, keeping them consistent by using a short list of common verbs and nouns.

You may also discover new participants or activities during this exercise. If so, simply go back and add them to the list. Then, revisit the new participant or activity using the previous modeling steps before moving on to the next modeling step.

**Step 4: Create and Group API Methods**

Once all activity steps have been identified, you will begin to see groups emerging around common usage patterns. Each of your step names can become an API method, grouped together by the type of activity or another kind of grouping. These groups of steps will become APIs and the steps themselves will become API methods.

For those somewhat familiar with REST APIs already, we will not be mapping these methods into specific HTTP methods and response codes as that will happen in the design phase. The goal of this step is to simply capture the APIs and methods to validate your API requirements before moving into the more detailed process of API design.

Begin by looking at your activities and activity steps, looking for common concepts. In our previous example above, we find that we need an Items API, Cart API, an Orders API, and a Customers API. Let’s capture these APIs, a short description, and their methods:

**Items API**

The Items API will manage what is available from the store:

<table>
<thead>
<tr>
<th>Method</th>
<th>Participant(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Items</td>
<td>Customer</td>
<td>List/search for store items</td>
</tr>
</tbody>
</table>

**Cart API**

The Cart API will manage a user’s cart and the checkout process:

<table>
<thead>
<tr>
<th>Method</th>
<th>Participant(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Item To Cart</td>
<td>Customer</td>
<td>Add an item to the customer’s cart</td>
</tr>
<tr>
<td>Remove Item From Cart</td>
<td>Customer</td>
<td>Remove an item from the customer’s cart</td>
</tr>
<tr>
<td>Clear Cart</td>
<td>Customer</td>
<td>Remove all items from the customer’s cart</td>
</tr>
<tr>
<td>View Cart</td>
<td>Customer</td>
<td>View the current cart and total</td>
</tr>
<tr>
<td>Checkout</td>
<td>Customer</td>
<td>Create an order from the contents of the cart</td>
</tr>
</tbody>
</table>
Orders API

The Orders API will manage the user’s orders and allow customer support staff and managers to view order status:

<table>
<thead>
<tr>
<th>Method</th>
<th>Participant(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Orders For Customer</td>
<td>Customer, Manager, Customer Support</td>
<td>List a customer’s orders from most recent</td>
</tr>
<tr>
<td>Obtain order status</td>
<td>Customer, Manager, Customer Support</td>
<td>View the status of an order</td>
</tr>
</tbody>
</table>

Customers API

The Customers API will allow customer support staff and managers to lookup a customer by name, prior to viewing an order status:

<table>
<thead>
<tr>
<th>Method</th>
<th>Participant(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locate Customer</td>
<td>Manager, Customer Support</td>
<td>Find a customer by name, ID, phone</td>
</tr>
</tbody>
</table>

You will notice that many of the activity steps mapped directly into our API. That is intentional, as your API should represent the actions that your participants take to perform activities. While our example is pretty simple, some APIs may look a little different than the identified activity steps. However, it should not deviate too far from the original steps, otherwise you may be missing out on APIs to support the needs of your participants.

Should you encounter repeating activity steps, you will need to decide if each step belongs to a single API, or if the step is just a common pattern but belongs on each corresponding API. The general rule is to assign the step as a method on a single API only if it is a step that can be shared between activities with only the context of the API it belongs. For example, we may not want to offer the Locate Customer API method on the Orders API, even though the activity may require that we do so before looking up an order. The Customer really belongs to an API that manages customers, not to an API that is responsible for Orders. Those familiar with object oriented development will recognize this as the Single Responsibility Principle of the SOLID principles.

Once you have completed a first pass of your API model, you can move to the final step: validating your API model.

Step 5: Validate the API By Testing

The final step in modeling your API is to validate it. Like a good quality assurance (QA) team, your job is to ensure that your API will meet the requirements of everyone using it. That is why we begin the modeling process by identifying the participants. The participants are the ones that will be using your API, either directly or indirectly via someone else’s application.

There are three ways to validate your APIs:
1. Walk through each activity step and identify the API method that will be invoked, and by whom. This is an informal way to verify that your APIs match your original activities and activity steps.

2. Create use cases and validate that your API can realize each one. This is useful if you have business analysts or subject matter experts capable of writing use cases.

3. Build workflow or interaction diagrams that demonstrate typical usage scenarios. This is common for more technical teams, or for teams that have business process experience.

As you validate your APIs, look for methods that are missing. You may also want to make notes about APIs that have dependencies on other APIs or that may experience heavy usage. While not necessary, this may guide some of your decisions as you move into the design and development phases.

**Common Questions About API Modeling**

API modeling is meant to be a simple exercise that is both approachable and extremely useful prior to design and development. Below are some common questions related to API modeling.

**Do You Need the UI Built Before You Model?**

A user interface does not need to be available before you model your API. However, if one of the goals of the API is to support a mobile and/or web application, it is very helpful to have some or all of the user interface mocked up prior to API design. This ensures that all activities can be easily identified during API modeling.

**What Kind Of Apps Or Tools Exist For API Modeling?**

Typically, a spreadsheet or document is sufficient to capture the API model. A collaborative spreadsheet, such as the kind offered by Google Docs, can allow a team to capture and refine the API model in realtime without the need to email changes between team members. As of this writing, we are not aware of any other applications available that specifically target API modeling.

**How Do I Facilitate API Modeling Within A Department Or Small Team?**

There are two common ways to manage collaborative API modeling: expert driven modeling and team driven modeling. Expert driven modeling starts with subject matter experts (SMEs), who then present to a team for review and feedback.

Team driven modeling involves a collaborative effort, generally around a table or whiteboard, and requires a moderator. The moderator’s job is to move the team through each activity, documenting the output of each step, and gathering everything into a shared document after the modeling exercise has completed.
Team modeling may go through all five steps, or the team may decide to allow a SME to start with the first two or three steps and then allow the team to gather to complete the remaining steps together. This decision should be based on the number of subject matter experts and team availability.

**Next Step: Design**

API modeling is a useful technique and leads to a well-designed API. Remember, you may find that you need to go through the process a few times as you uncover new requirements. You may also need to include other stakeholders or subject matter experts if you discover roadblocks in your domain knowledge. API modeling is often a continuous process that precedes the design of any new API, even for an API already in production.

Once you have completed your API model, you are ready to design your API. The API design phase will apply the pragmatic principles of RESTful API design, while meeting the business and user requirements of your API. Of course, before we apply the principles, we need to make sure we cover them. That will be our focus for the next chapter.
Chapter 5 - The Principles of HTTP and REST

Now that we have a model, most people’s instinct is to start writing code and implementing the API, but before we dive into the details of how to actually build the API, we need to understand the medium we are using. That medium is the HTTP protocol. By using an established protocol, we can benefit from the work of thousands of others, applying their understanding and approaches to problems that we face. Once we get into the details of the protocol, you will see there are features and functionality that covers all of your most common cases and even some of your most unique scenarios. To begin this exploration, we need to understand how applications communicate.

HTTP Core Concepts

Headers and Payloads

Every HTTP request and response is made of two parts: the headers and the payload. The payload is familiar to anyone who browses the web. In those cases, the payload will be simple HTML displayed to the user. When we are working with APIs, the payload will normally be JSON or XML and will be processed by a client application. While the payload is important to using an API, quite often the headers that you don’t see are even more important.

In practice, below is an example of some of the headers returned by the Github API:

HTTP/1.1 200 OK
Server: GitHub.com
Date: Thu, 07 Aug 2014 02:52:34 GMT
Content-Type: application/json; charset=utf-8
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 58
X-RateLimit-Reset: 1407382965
ETag: "ac2df149d98a36e7618793118dc0c752"
X-GitHub-Media-Type: github.v3
Content-Security-Policy: default-src 'none'
Content-Length: 1894
X-GitHub-Request-Id: 6B1045A4:1E51:590DF41:53E2E9F2
X-Content-Type-Options: nosniff
Vary: Accept-Encoding
X-Served-By: 3061975e1f37121b3751604ad153c687
Unfortunately, we don’t usually see the headers in most requests or responses, we just see the payload. In order to get access to the headers, we need to do one of a couple different things. If you’re working in Chrome or Firefox, check out the Live HTTP Headers Add On. You can download it from the Chrome Store or the Firefox Addon Repository. It’s the fastest and easiest way to see the headers for each request. If you prefer the command line, using the `-I` command with curl works effectively the same. Alternatively, if you want all the power of curl along with a user friendly interface, Postman is your answer.

We get a number of simple headers such as the datetime of when the payload was generated and even the domain of the request. The more advanced headers such as Content-Security-Policy and others help define how to interact with the API. Another type of header are those that begin with `X-`. Technically, these are deprecated and not recommended but there are still many APIs in the wild that actively use this approach. Often the most important is the Status code within the message. This is called an HTTP Response Code.

**HTTP Response Codes**

Response codes serve as the core of the HTTP protocol. They are used to express various success and failure states in addition to a variety of other things that occur during normal communications. There are five classes of responses defined in RFC 2616: HTTP 1.1: Informational (1xx), Success (2xx), Redirection (3xx), Client Error (4xx) and Server Error (5xx). Or in more colloquial terms: informational, success, moved over there, you screwed up, I screwed up.

There are over 40 individual response codes but in API design and usage, these are the most common:

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>The request has succeeded</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
<td>The request has been fulfilled and resulted in a new resource being created.</td>
</tr>
<tr>
<td>202</td>
<td>Accepted</td>
<td>The request has been accepted for processing, but the processing has not been completed.</td>
</tr>
<tr>
<td>204</td>
<td>No Content</td>
<td>The server has fulfilled the request but does not need to return a body. This is common for delete operations.</td>
</tr>
<tr>
<td>301</td>
<td>Moved Permanently</td>
<td>The resource has been assigned a new permanent URI.</td>
</tr>
<tr>
<td>302</td>
<td>Moved Temporarily</td>
<td>The resource is temporarily available at a new URI.</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>The request could not be understood by the server due to malformed syntax.</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td>The request requires user authentication.</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
<td>The server understood the request, but is refusing to fulfill it.</td>
</tr>
</tbody>
</table>

²¹https://chrome.google.com/webstore/detail/live-http-headers/iaiioopjcekamldfgehbdclcnpgnlo?hl=en
²³http://www.getpostman.com/
²⁴http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>404</td>
<td>Not Found</td>
<td>The server has not found anything matching the requested URI.</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
<td>The server encountered an unexpected condition which prevented it from fulfilling the request.</td>
</tr>
</tbody>
</table>

Despite proper usage of these codes being easy and well-established, some web applications have misused and abused them causing confusion at best and frustration at worst. For example, many content management systems used to return a 200 OK even if the requested content item was not found or otherwise not available. Most likely a 404 Not Found or 403 Forbidden would make more sense in these scenarios.

In many scenarios in your own API design, your first impulse will be to create additional response codes outside of those defined in RFC 2616. Don't. You should instead focus on using the existing response codes properly and consistently. We will talk about how to responsibly and effectively extend these codes later with real world examples. This principle bears repeating:

> Do not create new response codes or change the meaning of the standard HTTP response codes. If you want to provide more information or context when an error occurs, vnd.error is a powerful approach which is steadily maturing.

### Content Negotiation

Content Negotiation is one of the most powerful yet underutilized aspects of APIs. To be frank, it is a somewhat complicated concept but doing it properly opens some useful options. Here is the formal definition:

> Content negotiation is a mechanism defined in the HTTP specification that makes it possible to serve different versions of a document (or more generally, a resource representation) at the same URI, so that user agents can specify which version fit their capabilities the best.

[Wikipedia Definition of Content Negotiation](http://en.wikipedia.org/wiki/Content_negotiation)

At first glance, this seems complicated but if you have traveled internationally, you may have encountered a more tangible example. When you approach a airline ticket agent or clerk at a tourist location, they will often greet you in multiple languages. For example, in Quebec, they may say hello in both English and French. Depending on how you respond will determine which language they continue speaking. It is content negotiation in the real world.

In practice, content negotiation between APIs works much the same way. When the client makes a request, it can include headers like `Accept` or `Accept-Language` to specify how the systems will communicate. Such a request looks like this:

25[http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)
26[https://github.com/blongden/vnd.error](https://github.com/blongden/vnd.error)
Accept-Language: de

In this case, the de specifies that the server should respond in German. Alternatively, the client can specify a series of languages to communicate and rank them according to preference. We use the quality or “q” score to define this preference. It is simply a value between 1.0 (most important) to 0.0 (least important) that allows the client to state this preference. In practice, that looks like this:

Accept-Language: de; q=1.0, en; q=0.5

In this case, the client is requesting German and if it is not available English is an acceptable response.

Accept: text/html; q=1.0, text/*; q=0.8, image/jpeg; q=0.6, */*; q=0.1

In the Accept header, the file type specification works exactly the same. In this request, the preference is for HTML. If that is not available, some form of text is the next option, then various image are acceptable but less so. Finally, if nothing else is possible, accept anything (/) just to provide something to the user.

In the scenario where the client and server cannot reach an agreement on what media type to use, the server can return a 415 Unsupported Media Type response code. This allows both the client and the server to understand what occurred and potentially negotiate content in another way.

**HTTP Verbs**

In the HTTP specification, it defines a set of methods or verbs that we use to accomplish all of our desired actions and interactions. While there are numerous verbs, there are five commonly used for APIs. Each of these verbs have specific patterns and actions associated with them as described here:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieves a specified collection or resource</td>
</tr>
<tr>
<td>POST</td>
<td>Creates a resource as described or modifies the underlying state of the system, often used in processing</td>
</tr>
<tr>
<td>PUT</td>
<td>Updates a resource at the specified URI</td>
</tr>
<tr>
<td>PATCH</td>
<td>Updates a partial resource at the specified URI</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes the specified resource</td>
</tr>
</tbody>
</table>

These verbs form the foundation of most APIs. They create a common language and pattern for developers to use to interact with APIs and understand their functionality. You may find scenarios where you would want to create additional verbs. **Don’t.** Most HTTP client libraries cannot support arbitrary verbs. Therefore verbs not formally defined in the specifications my not be available to some users of some libraries some of the time.

Fundamentally, once you understand verbs, you have gone a long way towards successfully understanding APIs as a whole. Most well-designed APIs will use the verbs consistently and the same as any other API. As a result, the only aspect that changes from API to API are the nouns or
resources. As our next step, we will look at these concepts and see how they fit together to build useful and powerful APIs.

**Web APIs and the Richardson Maturity Model**

To understand how these concepts fit together to build RESTful APIs, we need to take a step back and look at what each of them brings to the table. We will do this with the Richardson Maturity Model\(^{28}\) as described by Martin Fowler, author of Patterns of Enterprise Application Architecture. The Richardson Maturity Model, or RMM for short, is a simple model that describes the four levels in how APIs fit together and how the concepts build on one another.

While some of the concepts described in the RMM are better defined and therefore more widely established, there are implementations of all the concepts being used in the real world. Examples of each are covered throughout this book.

---

**Level 0: “The Swamp of POX” or Remote Procedure Invocation**

In this sort of architecture, you have a single API Endpoint which receives all of your requests and processing instructions. Further, the name of the desired action is reflected with some sort of action parameter or even embedded within the payload of the request. You may have seen this from APIs with methods like `getName` or `processResult`. In those cases, the receiving system can only make

---

(\(^{28}\)http://martinfowler.com/articles/richardsonMaturityModel.html)
sense of the request once the entire message is parsed. The payload itself will often be a vendor-specific flavor of XML but could be JSON or a binary format just as easily.

While this works in concept, it tends towards duplication and obfuscation. Since it tends to only use GET and POST instead of the four HTTP verbs, we lose granularity on the type of request being made. Since there are no nouns or resources, we lose understanding of what items are being interacted with. Since the only response codes used are generally 200, 404, or 500, we lose insight into the results of our request. And finally since the payload is opaque, the related resources and additional information is often obfuscated. While this architecture is a starting point, there is a great deal of room for improvement.

**Level 1: Adding Resources**

The first step towards building a modern web API is using nouns. Instead of interacting with a single API Endpoint, we have separate endpoints for each noun or resource in the system. If we’re working with a project management system, we might have an endpoint for projects and another for tasks. For the object oriented developers out there, this will feel much more familiar. The further benefit of this architecture is that it allows us to interact with the individual resources by name. Instead of using `getProject`, we have something that looks like `/projects/1234`. We gain some understanding of the API and how it’s being used simply by the URLs that are being called. It’s not perfect but it’s a starting point.

**Level 2: Adding the HTTP Verbs and Response Codes**

Now that we’ve redefined what we’re interacting with, at the next level we can redefine how we’re working with those resources. This is where we start using the power of the underlying medium of the internet: HTTP and its verbs. Instead of simply using GET and POST to retrieve items, perform searches, and update data, we use the full range of verbs - GET, PUT, POST, and DELETE - to undertake various actions.

In the last section we learned about what the verbs do, so by making use of them, we gain yet another insight in the transaction from simply looking at the API call. When we see a request such as `DELETE /projects/1234` or `GET /users/` we can immediately tell what is happening without looking deeper into the request or even the parameters. More importantly, we’re not duplicating information like a `GET /getProjects` request would.

Further, by implementing the HTTP Response Codes, we introduce another level of granularity that wasn’t available before. Instead of receiving a generic 200 response code regardless of the action, we can get back codes such as `201 Created` which tells us a resource was created. Even the mysterious `400 Bad Request` offers more understanding than simply always returning `200 OK`. More importantly, we have the opportunity to add extra information to describe the situation and the results of the request.

At this stage, we have nouns, verbs, and response codes in play. While we have gained a great deal of insight and understanding of both the request and the results of the request, we have not provided a way to navigate the system more fully. In order to do that, we need hypermedia.
Level 3: Introducing Hypermedia

Hypermedia is the pinnacle of the Richardson Maturity Model. It is driven by the concept that APIs should be self-descriptive to the point where the out of band (aka outside the API) documentation should be nearly optional. In practice, this looks like links with Link Relations describing how the links are related and used. More importantly, those links are context-sensitive and change based on where the user is within the API and what features and functionality are available.

For example, a link may be described as the “edit” relation which leads to a representation of a form to edit a resource. If the user does not have permission to edit, instead of supplying the link and returning a 403 Forbidden, a hypermedia representation of the API simply wouldn’t include the link. In this way, hypermedia can be used to drive user interfaces by showing or hiding buttons based on availability of options.

Hypermedia itself is still a contentious topic. It elicits enthusiasm and glee from one side while generating disdain and annoyance on the other. While the usefulness of hypermedia is still under debate, the definition and intended usage is generally agreed upon. We’ll cover this in much greater detail in the following section on Uniform Interfaces.

Putting It Together: What We Recommend

When we combine all the concepts of the Richardson Maturity Model, we create APIs that are descriptive, powerful, and overall more predictable in their use and architecture. We create a scenario where we already understand the verbs involved so the only new or different aspect from API to API are the nouns themselves. Further, with the capabilities that hypermedia hints at, we don’t have to rely on potentially outdated documentation to explore an API. The API itself tells us what can be accomplished and how.

<table>
<thead>
<tr>
<th>RMM Level</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>Single URI, Single Verb, No Links (SOAP, XML-RPC)</td>
</tr>
<tr>
<td>Level 1</td>
<td>Many URIs, Single Verb, No Links</td>
</tr>
<tr>
<td>Level 2</td>
<td>Many URIs, Many Verbs, No Links</td>
</tr>
<tr>
<td>Level 3</td>
<td>Many URIs, Many Verbs, Links</td>
</tr>
</tbody>
</table>

Unfortunately, many APIs are simple XML over HTTP and begin and end at Level 0. Most CRUD-type APIs will be designed to take advantage of unique nouns and occasionally the verbs, so they will be Level 1 at minimum, Level 2 at best. For the rest of this book, we will focus exclusively on Level 2 and above. While we believe that hypermedia still needs some work, we firmly believe that it is useful and will only become more so in the coming months and years.
Back to Basics: The 6 REST Constraints

It is worth remembering that while REST APIs use the HTTP protocol, REST itself is not a protocol. It is simply a set of constraints and agreements on how the individual components should work together. This is both good and bad. It is good in that implementers have flexibility to address their issues in the manner they see fit according to their particular requirements and the design trade-offs they choose. Unfortunately, it is bad in that it causes disagreements on what is “RESTful” and if something is “REST enough.” Frankly, these disagreements are beside the point but they have occurred regularly in the API community for years.

The difference between SOAP and REST is best described by an analogy. In terms of implementation and documentation, SOAP is like a mortgage while REST is like borrowing $10 from a friend. In SOAP a large amount of documentation is required to begin the transaction. Further, everything is very well defined including how to start the transaction, what completion looks like, and how to resolve the various issues and problems which may occur along the way. As a result, you have a heavily defined scenario with little room for flexibility but generally inspires confidence in the process and results. Alternatively, if you borrow money from a friend for lunch, there is less process involved and more flexible. You may simply return the money tomorrow or maybe buy them a comparable lunch or maybe buy drinks at the next happy hour or maybe they do not expect the money back or maybe you do not pay them back at all. The flexibility is great but if you’re not careful, it puts some ambiguity into the mix. This ambiguity must be figured out and resolved.

As a general set of principles, the six REST constraints serve to establish expectations and create some structure and agreement around that flexibility. These are not hard and fast rules but should be considered guidelines and best practices for implementation. Like any best practices, you can choose not to follow them but it should be done only after careful consideration of the consequences.

Client/Server

The first constraint is simply that an API should be designed to be used in a Client/Server architecture. As in there should be a Server which hosts the data and some portion of the business logic and a Client which retrieves data and interacts with it in some way. In most cases, there will be business logic on both sides. The alternative is a single component system without the ability to upgrade or vary the pieces.

There are numerous benefits to this setup. The first and foremost is that it allows you to vary each side independently of the other. In web development, we do this all the time. Our web and database servers operate independent of one another and can therefore be upgraded and managed separately. We have the same situation when we consider web browsing. When we use a system like Gmail, we can choose to use any browser we prefer and Google can upgrade Gmail whenever they wish usually without requiring our browser to change.

Stateless

The second constraint is that an API should be Stateless. This allows each and every API request to stand alone and be considered and processed independently of one another in any order required.
In practical terms, this usually implies sending credentials with each and every request because we cannot count on the "log on" request getting there before the subsequent requests.

Here is an example of what it might look like:

curl -X POST 'https://api.twilio.com/2010-04-01/Accounts/ACxxxx/SMS/Messages.xml'
  -d 'From=%2B15125551212'
  -d 'To=7035551212'
  -d 'Body=Never+gonna+give+you+up+never+gonna+let+you+down.'
  -u ACxxxx:{AuthToken}

The key part is the -u parameter which explicitly specifies the credentials. The rest of the parameters are used to add data values to the request.

Alternatively, most web-based systems will use some sort of session or cookie to keep track of being logged in, user preferences or similar aspects. While this works for smaller operations, it does not work at scale. Specifically because sessions are machine-based by default and the easiest way to handle more traffic is to add more servers. There are solutions to synchronize sessions across multiple machines or otherwise have a shared session store but this just introduces a different class of problems which require additional solutions.

The ability to process requests in any order is a huge strength for an API. For example, since requests are independent, we can simply add more servers when the load starts growing beyond our original planned capacity. This gives us a much more flexible system as we can add support for what we need when we need it. We don’t have to predict every possible scenario before we launch the system.

Finally, if all else fails, and we lose a request, all later requests can still work. Or to put it another way, just because your first request failed doesn’t mean that all of your others did too. This makes our systems fault tolerant and more stable and reliable in the case of errors.

**Cacheable**

The next constraint is cacheability. It requires that each message includes information identifying whether it is cacheable or not. The primary benefit of well-planned caching is to improve network and application usage by reducing - or in some cases eliminating - requests. This makes the system more reliable and scalable. After all, the fastest request is the one you do not have to perform.

When we consider APIs specifically, we encounter the concept of Idempotency or the idea that an operation can occur multiple times and the same result occurs on the N+1 time as it did the first. Of the four key HTTP methods - GET, PUT, DELETE and POST - the first three are idempotent. Since the underlying system should not change on subsequent requests, the results of those commands should be cacheable to the client systems. The last method (POST) is used to create new resources and/or change the status of existing resources, so it is not considered cacheable.

In practice, cache control is accomplished through concepts like HTTP ETags\(^{29}\) which are generated whenever a given piece of content changes. When the client makes the next request,
it includes the ETag value. If the API responds with a 304 Not Modified response code, the client can simply use the results of the previous request. This can greatly reduce network traffic which is especially important in sometimes-offline environments common in mobile.

**Layered System**

The next constraint is a Layered System and means that your Client should not be built on the assumption it is communicating directly with the Server. Or in other words, there can be layers or systems in between the two for caching and a variety of other components. The API management companies - Mashery, 3Scale and others - take advantage of this aspect to insert logging, access control, and load balancing between you and the actual API.

In daily use, we use this for domain names. For example, we could access individual websites using their IP addresses directly. It is not practical but it would work. Luckily, our Domain Name Service takes an easily remembered domain like google.com and translates it to a hard to remember IP address. Then your browser uses the IP address to retrieve the contents of the website.

**Code on Demand (optional)**

Code on Demand is the most powerful but - until recently - underutilized constraints. The concept is that when a Client requests a resource, it also receives code to act upon it. The real power is that the Client does not have to know what is in the code, it just has to understand how to execute it. The primary benefit is that the API can grow and extend itself without requiring the Client applications to upgrade. We can get new functionality for “free.”

While executing arbitrary code sounds like a security nightmare, it is something we do every day with Javascript. For example, when you load an application such as Gmail, your browser also retrieves the code required to interact with that page. It does not need to know what is in the Javascript, it just knows how to execute it. More importantly, as new features and functionality become available, they are immediately available to the user without requiring a browser upgrade. It is how we take a relatively dumb piece of software (a browser) and make it a one-size-fits-all tool for our email, CRMs, ERP, blogs, and every other web application out there.

**Uniform Interfaces**

Uniform Interfaces is the final constraint and the most complex of all. In simplest terms, the idea is that you should be able to interact with a resource the same way you interact with all the other resources. If you view a given resource in a particular way, all the resources should be viewable the same way. This does not necessarily mean that every resource has all the same interactions as every other, but the ones that are available should operate the same. To drive this home, we will break this into a few pieces.

**Identification of Resources**

The first components is Identification of Resources. Each resource should be uniquely addressable via a particular URI. Generally, there will be one and only one way to access this resource but that
is not absolutely required. Within a RESTful API, generally you would expect to see /noun/:id or /noun/:id/action instead of /?n=noun&id=:id or /?n=noun&a=action&id=:id

Technically there is no hard and fast rule here so the second set of URIs are acceptable too. They are not as pretty - which is a minor inconvenience - but more importantly, they just look incorrect. If you’re building an API and have the choice between the two types of URIs, please choose the first. They are generally better accepted and less likely to cause people to stumble as they use your API.

Manipulation of those Resources

Next, we need to be able to manipulate or interact with the Resources through those representations. Every interaction with a given resource should happen through the name/identification that we already gave it.

In the real world, here is an example using the Twilio API:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Resource</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/2010-04-01/Accounts/{AccountSid}/Calls/:id</td>
<td>Retreive the information on the specified Call</td>
</tr>
<tr>
<td>POST</td>
<td>/2010-04-01/Accounts/{AccountSid}/Calls</td>
<td>Create a new Call</td>
</tr>
<tr>
<td>GET</td>
<td>/2010-04-01/Accounts/{AccountSid}/SMS/:id</td>
<td>Retreive the information on an SMS</td>
</tr>
<tr>
<td>POST</td>
<td>/2010-04-01/Accounts/{AccountSid}/SMS</td>
<td>Create a new SMS</td>
</tr>
<tr>
<td>DELETE</td>
<td>/2010-04-01/Accounts/{AccountSid}/Recordings</td>
<td>Delete the specified Recording</td>
</tr>
</tbody>
</table>

There are three resources (or Nouns) listed here - Calls, Recordings, and SMS messages - and they each have verbs associated with specific actions. More importantly, the only way to interact with these nouns is via these verbs. By using these patterns, there is a consistent and predictable way to interact with every component of the system. This helps us document things more easily and helps developers who are learning about the system.

Self-descriptive Messages

The next principle of uniform interfaces is the concept of Self-descriptive messages. This is important because by making messages stand alone with both its own caching information and processing details, we can create and use different types of messages very simply.

Since the messages tell us how to process themselves, our client only needs to know how to retrieve and execute those instructions, not how to generate them directly. Again, HTML and Javascript are good examples of this. When you make a request to gmail.com, the payload itself has instructions on which Javascript files to include. Your browser simply includes the files and calls the correct function when the payload says to. Further, we don’t have to figure out how to view another resource such as an email. The payload itself tells us how to do that.

This is another area of debate within the API community. There are a number of competing
standards - Atom³⁰, HAL³¹, Collection JSON³², Siren³³, UBER³⁴ and others - which describe a message in different ways. While Atom and HAL are the most common, each has its own strengths and weaknesses along with supporting toolsets. There is no clear winner in this space at this time.

**Hypermedia as the Engine of Application State (HATEOAS)**

The final principle turns the API payloads into tools which drive the state and availability of features and functionality within the system. The format definition is:

> Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g. by hyperlinks within hypertext). Except for simple fixed entry points to the application, a client does not assume that any particular actions will be available for any particular resources beyond those described in representations.

_Wikipedia Definition of HATEOAS³⁵_

Or in less formal terms, it is a Choose Your Own Adventure Book. If you’ve never read a choose your own adventure book, you don’t read the pages in consecutive order: 1, 2, 3, 4. Instead, you start on page 1 and every time your character is faced with a choice, you have the options to turn to alternate pages depending on your choice. If you have two doors, opening the right door might send you to page 10 while the left door sends you to page 20.. and in both cases, you’re faced with another choice.

This is how hypermedia works. At every point of the API, you have different things you can do with the API and each different thing is referenced by a different link, representing your choices. This allows you to make more intelligent decisions based on the options you actually have as opposed to the options you might have in some scenarios.

In practice, this is what it looks like when you make a request to the root of the Github API:

```json
{
  "current_user_url": "https://api.github.com/user",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url": "https://api.github.com/search/code?q={query}&page,per_page,sort,order",
  "emails_url": "https://api.github.com/user/emails",
  "emojis_url": "https://api.github.com/emojis",
  "events_url": "https://api.github.com/events",
  "feeds_url": "https://api.github.com/feeds",
  "following_url": "https://api.github.com/user/following{/target}",
  "gists_url": "https://api.github.com/gists/{gist_id}"
}
```

³¹ [http://stateless.co/hal_specification.html](http://stateless.co/hal_specification.html)
³³ [https://github.com/kevinswiber/siren](https://github.com/kevinswiber/siren)
Once you make a request to any of those resources, you get another list of related resources which allows you to traverse the API. Because the API itself provides these paths, your application does not have to store or determine any of these URIs in advance. Our applications just have to remember the name of the URI instead.

If you think about it, that’s how humans work too. We don’t remember what the Log On URL is for amazon.com. Instead, we go to amazon.com and skim the page until we see a button labeled “Sign In” or “Log On” or something similar. Also, we don’t have to memorize the link to get to the books section of the site. Again we visit the homepage and skim for a link labeled “Books”.

When our client applications don’t have to remember the specific URLs for each and every action or link we want to visit, our clients can be simpler and more flexible. More importantly, as your API adds additional functionality, you can just add it and our clients can update automagically.

The Next Steps

In our previous chapter, we created a model for our API from a few initial stories. At the time, we kept the discussion in terms of business goals and tangible steps that our participants complete. Now
that we have covered the core concepts of HTTP and REST, the next step is to put the ingredients together. We will start by identifying our resources, combine them with the HTTP verbs, and start mapping the actions to specific response codes.
Chapter 6 - From Modeling to RESTful Design

Now that you have modeled your API and have a good understanding of the principles of REST, it is time to start the API design process. While the goal of the API modeling phase is to explore and capture the API requirements, API design maps those requirements to the RESTful principles that we learned about in the previous chapter. This is where the fun really begins, both for you as the designer and for the developers that will eventually consume the API.

As you move from the modeling to the design phase, you will be faced with a variety of decisions. Some of these will resolve easily, while others will take time and deliberation. Just know that it is difficult to get your API design right the first time. This is why we encourage you to spend time designing and prototyping your API before you start implementing it. It will save you time, money and stress.

This chapter outlines recommended steps to designing a great API. However, you may encounter some confusion or uncertainty about common patterns and practices regarding REST APIs. We encourage you to review our coming appendix on API design patterns for a more detailed exploration of API design details and patterns to help clarify these kinds of issues.

Building Your Resource Taxonomy

A taxonomy is simply a classification of concepts. An API taxonomy captures the set of resources you will be offering, and their relationships to other resources.

If you have already begun modeling your API, you likely have the beginnings of a taxonomy already. If necessary, stop designing and go back to Chapter 4 to fully model your API first to ensure that you have a complete understanding of what you will be designing.

To get started building your taxonomy, make a list of the resources you have modeled already. These are often identified as the nouns in your system. Keep in mind that some resources may belong as a child collection of another resource. We’ll deal with that aspect soon, but for now just focus on listing all of the resources.

Building a Taxonomy For Our Example

Let’s review the resources that we modeled in Chapter 4 to build an example taxonomy:

<table>
<thead>
<tr>
<th>Resource Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
</tr>
<tr>
<td>Cart</td>
</tr>
<tr>
<td>Orders</td>
</tr>
<tr>
<td>Customers</td>
</tr>
</tbody>
</table>
Pretty simple so far, right? Then let’s talk about how to handle relationships between resources.

**How to Handle Resource Relationships and Composition**

Rarely do APIs offer only flat resources without relationships to one another. Instead, resources often contain child resources or reference other resources. In these cases, our API design must ensure that we can interact with the top-level resource as well as nested and related resources.

To better design our API, we need to understand the three types of resource relationships:

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>The resources exist stand alone without the other’s existence, but may reference each other.</td>
</tr>
<tr>
<td>Dependent</td>
<td>One resource cannot exist without the existence of the parent resource.</td>
</tr>
<tr>
<td>Associative</td>
<td>The resources exist independently, however their relationship contains or requires additional properties to describe it.</td>
</tr>
</tbody>
</table>

How you design your API around related resources can be determined based on a few questions. The answers to these questions will determine if your resources are independent, dependent, or associative:

1. Can both resources exist without the other? Then we have an **Independent** resource.
2. Does one resource exist only when the other exists? Then we have a **Dependent** resource.
3. Does the relationship between resources require more state than just the links between them? Then we have an **Associative** resource.

Resources with independent relationships exist separately and reference each other via Hypermedia links as necessary.

As an example, we may have two API endpoints to provide the details of a movie and actor:

GET /movies/:movie_id

GET /actors/:actor_id

However, actors in a movie may portray one or more characters. The characters do not exist outside the context of an actor and a movie. Therefore, this is a dependent relationship between resources as we have no need to offer the characters resource as a top level resource:
GET /actors/:actor_id/characters
GET /actors/:actor_id/characters/:characters_id
GET /movies/:movie_id/characters
GET /movies/:movie_id/characters/:character_id

But, what if we want to track the relationships between characters? Then we may be better off creating an associative resource, where they can be viewed and tracked separately:
GET /characters/:character_id

Taking this approach will allow us to lookup a character to see both the actor(s) that portrayed the character and the movie(s) that the character appeared. This is only possible (without a lot of heavy lifting by the API client) by changing the relationship between actors and movies into a third resource that can be directly accessed.

Capturing Relationships To Our Example

Returning to our example API from Chapter 4, let’s capture the relationships alongside our resources to help us look for nested resources and for linking with our representations later in the chapter.

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Resource Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>-</td>
</tr>
<tr>
<td>Cart</td>
<td>has Items</td>
</tr>
<tr>
<td>Cart Items</td>
<td>belongs to a Cart, has Items</td>
</tr>
<tr>
<td>Orders</td>
<td>has Items</td>
</tr>
<tr>
<td>Customers</td>
<td>has Orders</td>
</tr>
</tbody>
</table>

Our Cart API will likely require a nested resource collection to support adding, updating, and removing individual items from the cart. So, where would Cart Items belong in our taxonomy? Since Cart Items belong to a Cart and cannot exist outside of it, it will be a dependent resource and therefore is modeled as a child collection:

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Related Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>-</td>
</tr>
<tr>
<td>Cart</td>
<td>has Items</td>
</tr>
<tr>
<td>Cart Items</td>
<td>belongs to a Cart, has Items</td>
</tr>
<tr>
<td>Orders</td>
<td>has Items</td>
</tr>
<tr>
<td>Customers</td>
<td>has Orders</td>
</tr>
</tbody>
</table>

Understanding the relationships between your resources is important as you build out your resource taxonomy. We will discuss resource linking later in this chapter.
Defining Resource Lifecycles

Once we have identified our RESTful API resources, it is time to start mapping them to our HTTP verbs. In Chapter 4, we were able to capture most activities using a specific list of actions: “list”, “clear”, “view”, etc. Depending on the verbs you commonly use during modeling, you will likely find that certain ones can be mapped directly to REST patterns. Below is a table that captures common words used during modeling that often directly map to our HTTP verbs:

<table>
<thead>
<tr>
<th>Modeling Actions</th>
<th>Typical HTTP Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>“List”, “Search”, “Match”, “View All”</td>
<td>GET resource collection</td>
</tr>
<tr>
<td>“Show”, “Retrieve”, “View”</td>
<td>GET resource instance</td>
</tr>
<tr>
<td>“Create”, “Add”</td>
<td>POST create a new resource</td>
</tr>
<tr>
<td>“Replace”</td>
<td>PUT update a resource collection</td>
</tr>
<tr>
<td>“Update”</td>
<td>PUT update a resource instance</td>
</tr>
<tr>
<td>“Delete All”, “Remove All”, “Clear”, “Reset”</td>
<td>DELETE delete a resource collection</td>
</tr>
<tr>
<td>“Delete”, “Remove”</td>
<td>DELETE delete a resource instance</td>
</tr>
<tr>
<td>&lt;other verbs&gt;</td>
<td>POST custom action on a resource collection or instance</td>
</tr>
</tbody>
</table>

As you review each resource in your taxonomy, determine which HTTP verbs you will need. Some resources may need the full set of HTTP verbs to support the lifecycle. However, some may need less (e.g. you cannot update a resource after creation) and some may need more to support custom semantics that don’t fit the standard HTTP verbs. Let’s talk about that next.

Using our understanding of REST principles, we know that we can directly map the following actions into URL templates that we’ll eventually implement:

<table>
<thead>
<tr>
<th>Capability</th>
<th>HTTP Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>List resources</td>
<td>GET /{collection name}</td>
</tr>
<tr>
<td>Get a resource</td>
<td>GET /{collection name}/{id}</td>
</tr>
<tr>
<td>Create a resource</td>
<td>POST /{collection name}</td>
</tr>
<tr>
<td>Update a resource</td>
<td>PUT /{collection name}/{id}</td>
</tr>
<tr>
<td>Delete a resource</td>
<td>DELETE /{collection name}/{id}</td>
</tr>
<tr>
<td>Replace all resources</td>
<td>PUT /{collection name}</td>
</tr>
<tr>
<td>Delete all resources</td>
<td>DELETE /{collection name}</td>
</tr>
</tbody>
</table>
Resource Lifecycles For Our Example

Applying these principles to our modeling example from Chapter 4 may result in an API design such as the following:

Items API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /items</td>
<td>List all available items from the store</td>
</tr>
</tbody>
</table>

Cart API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /cart</td>
<td>List details about the cart, including total cost and all items/quantities in our cart</td>
</tr>
<tr>
<td>POST /cart/items</td>
<td>Add an item to the cart with a specific quantity</td>
</tr>
<tr>
<td>PUT /cart/items/{itemId}</td>
<td>Update an item quantity already in the cart</td>
</tr>
<tr>
<td>DELETE /cart/items</td>
<td>Clear the contents of the cart</td>
</tr>
<tr>
<td>DELETE /cart/items/{itemId}</td>
<td>Remove an item from the cart</td>
</tr>
</tbody>
</table>

Orders API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /orders</td>
<td>List recent orders for the current customer</td>
</tr>
<tr>
<td>GET /orders/{orderId}</td>
<td>Order details, including the current status</td>
</tr>
</tbody>
</table>

Customers API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /customers{?customerId,name}</td>
<td>List customers with optional filters</td>
</tr>
</tbody>
</table>

Mapping Response Codes For Success and Failure

While we hope that every request sent from a client will result in success, that is not always the case. Your resources will need to handle a variety of results that indicate both success and failure. Understanding the different types of response codes and how they are commonly used will help create consistency across your API. In addition, using the more appropriate and descriptive response codes will also help clients to determine success or failure (and to what extend) without requiring the response payload to be parsed immediately. This is most important in low-processor situations, such as with mobile devices that need to conserve battery by reducing compute requirements.

It is important that you understand the uses for each kind of response code, whether they indicate
a success or failure, and when to use them (and when not to). We have witnessed a variety of APIs that don’t do this, creating confusion and bugs in client code. This goes back to our core rule that we first mentioned in Chapter 5:

Don’t create new response codes or change the meaning of the standard HTTP response codes.

We recommend reviewing the HTTP Response Codes section of Chapter 5 if you are uncertain about any of these response codes.

### Common Response Codes For All Requests

There are a variety of response codes that are common to most requests, no matter what kind of request being issued:

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Recommended Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Request was successful</td>
<td>200 OK</td>
</tr>
<tr>
<td>Error</td>
<td>User not authenticated</td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td>Error</td>
<td>User not authorized for endpoint request, often due to missing security privileges</td>
<td>403 Forbidden</td>
</tr>
</tbody>
</table>

You will use these response codes often and for a variety of reasons.

### Common Error Response Codes For GET

Since a GET request for a collection or resource is straight-forward, there aren’t too many status codes that handle other conditions. If you find yourself using other status codes, it might be an indication that you are using a GET request for some other kind of processing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Recommended Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Resource not found</td>
<td>404 Not Found</td>
</tr>
<tr>
<td>Error</td>
<td>Filter parameters provided were not valid</td>
<td>400 Bad Request</td>
</tr>
</tbody>
</table>

### Common Response Codes For POST

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Recommended Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>A custom action or processing request was successful</td>
<td>200 OK</td>
</tr>
<tr>
<td>Success</td>
<td>Created resource successfully</td>
<td>201 Created</td>
</tr>
<tr>
<td>Success</td>
<td>Long-running request has been accepted</td>
<td>202 Accepted</td>
</tr>
</tbody>
</table>
Since a POST request may be used for custom actions, as well as for creating new collections, we may just use a 200 OK response for our response code. However, a 201 Created is more appropriate when creating a new resource to indicate that a new resource now exists.

### Common Error Response Codes For PUT

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Recommended Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Updated resource successfully</td>
<td>200 OK</td>
</tr>
<tr>
<td>Error</td>
<td>Fields were not valid or required</td>
<td>400 Bad Request</td>
</tr>
<tr>
<td></td>
<td>fields were missing</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>Resource not found</td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>

### Common Error Response Codes For DELETE

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Recommended Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Resource deleted</td>
<td>204 No Content</td>
</tr>
<tr>
<td>Error</td>
<td>Resource not found</td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>

### Response Codes For Our Example

As we look at each of our API endpoints, we can start to fill in some of the expected response codes based on how each API is intended to behave:

#### Items API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
<th>Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /items</td>
<td>List all available items from the store</td>
<td>200 OK</td>
</tr>
</tbody>
</table>

#### Cart API
### Endpoint Description Response Codes

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
<th>Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /cart</td>
<td>List details about the cart, including total cost and all items/quantities in our cart</td>
<td>200 OK</td>
</tr>
<tr>
<td>POST /cart/items</td>
<td>Add an item to the cart with a specific quantity</td>
<td>201 Created</td>
</tr>
<tr>
<td>PUT /cart/items/{itemId}</td>
<td>Update an item quantity already in the cart</td>
<td>200 OK, 401 Unauthorized, 404 Not found (item not found in cart)</td>
</tr>
<tr>
<td>DELETE /cart/items</td>
<td>Clear the contents of the cart</td>
<td>204 No Content, 401 Unauthorized</td>
</tr>
<tr>
<td>DELETE /cart/items/{itemId}</td>
<td>Remove an item from the cart</td>
<td>204 No Content, 401 Unauthorized, 404 Not found (item not found in cart)</td>
</tr>
</tbody>
</table>

### Orders API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
<th>Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /orders</td>
<td>List recent orders for the current customer (or by customer id if permitted)</td>
<td>200 OK, 404 Not Found (Customer not found by id)</td>
</tr>
<tr>
<td>GET /orders/{orderId}</td>
<td>Order details, including the current status</td>
<td>200 OK, 404 Not Found (Order not found by id)</td>
</tr>
</tbody>
</table>

### Customers API

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
<th>Response Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /customers</td>
<td>List customers with optional filters</td>
<td>200 OK, 404 Not Found (Customer not found by id)</td>
</tr>
</tbody>
</table>

### REST Beyond CRUD

So far, we have spent most of our time discussing REST and its resource-oriented approach to building web services. But, what happens when you have an action that doesn’t fit the normal
Create, Read, Update, Delete (CRUD) workflow? Can we still use REST to handle our custom actions as well? Absolutely!

Remember how we talked about your API being a way to express your business capabilities? If all you are offering is a very basic REST API wrapper around some database tables, then you are likely not building any kind of significant business capability into your API. Look for opportunities to think beyond your data structures to find those hidden resources that will help developers do significant work with very little effort.

A common struggle when moving to REST is that APIs often map to a database table in early architectures. There is no rule that says this must be the case. Resources are simply business entities that may be acted upon by an API client. These might map to a single database table, multiple normalized database tables, or a database view. Or, they may represent a process that may be represented by a variety of conditions, from rows in a database table to manual processes that must be performed and marked as complete before proceeding.

Think about that again: resources may be more conceptual in nature, modeling third-party systems or workflow processes. This is where your API can start to demonstrate high value as you begin to expose business capabilities as additional resources.

For example, an API that supports purchasing seats for a sporting event might require a time-limited registration process before the reserved seats were returned to the availability pool. The API could offer a resource to model the seat reservation workflow. This resource would remove the seats from the availability pool while the customer completes the purchase. If the time limit of the reservation is exceeded, the backend system would return the seats to the pool and mark the seat reservation as expired, preventing the customer from completing the purchase upon the next API call. While this resource isn’t something that is long-lived and may not be directly modeled as a separate database table, it provides a conceptual resource that can be understood and acted upon by an API consumer without the requirement to understand anything about the internals of the backend system. It does, however, help the API consumer understand the constraints of how to locate, temporarily reserve, and obtain (or release) available seats.

Expect your REST APIs to move beyond the standard CRUD-based approach. These robust APIs often combine resources for data access with custom actions and workflow support to provide a powerful developer experience. Just be sure that you aren’t creating custom actions when what you really need is a resource to manage the state and underlying behavior. We’ll discuss that in greater detail later.

**Expanding Resources Through Hypermedia Linking**

As you move from resource identification, it is time to explore the links (relationships) each resource will provide. Resource links are just like links within a web page - they describe the options available for navigation. Links may be used for a variety of reasons:

1. To inform clients about the actions available given the resource’s current state
2. Provide hints to actions that are authorized for the current user
3. Allow clients to explore resource relationships

When we apply our RESTful patterns of HTTP verbs to our link, we will know how request new or related resources (i.e. GET), create new resources (i.e. POST), and update or delete a resource (i.e. PUT and DELETE) without hard-coding our client with this knowledge.

Links have a minimum of two fields: the relationship type, or ‘rel’, and the href. The relationship type defines how the link is related to the resource. Examples include: self, previous, next, or a variety of actions. Some APIs offer an additional name or prompt field, allowing links to provide a description for visualization and human interaction within a user interface. Finally, a type or mediaType field may be offered to provide additional details for clients regarding the content type a link may be offering (e.g. a specific image or binary format).

When clients consume provided links, rather than trying to hard-code them, it allows the server to drive what navigation options and additional actions are available to the API consumer based on their current credentials.

For example, think of the Amazon.com homepage. Do you know the precise URL required to log in? Most likely not. More likely, you visually scan the page until you find a keyword like “log in.” Link relationships provide the same function. Instead of trying to hardcode every option, the client can simply look for links with the expected keywords (relations) to determine the next paths.

**Link Thyself**

Every resource should define a link to itself. While this seems to be redundant, or perhaps wasteful, there is a valid reason for returning a link to a returned resource: clients should never have to track the link that returned a resource separate from the resource representation itself. While tracking the url separately isn’t a difficult task necessarily, we want to ensure that clients don’t have to do any more work than necessary. In addition, returning a self link to the resource also provides more consistency within the response payload.

**Resource State Linking**

Next, look for any constraints and state transitions that a resource may require. These constraints should be surfaced as links, offering insight into what actions are allowed and not allowed given the current state of a resource. Often, these constraints emerge during the API modeling step from Chapter 4, but sometimes surfacing these constraints requires an additional effort during the design phase.

For example, if we have an Article resource for a content management system, we may have the following resource state transitions:

- Draft - being written, not published yet
- Submitted - publication requested, review required
- Approved - review approved, article is now marked for publication
- Declined - review not approved, article may be rewritten and re-submitted or deleted
- Published - article is now available for viewing
- Deleted - article has been marked as deleted and no longer available for viewing

State diagrams, such as the one shown below for our Article resource, is a great way to capture these constraints and ensure that the correct API links are returned for a resource and help testing the underlying implementation as well. Here is an example state diagram for this example:

![State Diagram](image)

A state diagram example for a content management system

From this one diagram, you will be able to apply the state transition and business rules within your API implementation, and also expose these rules as links from within your resource representation payloads. So, be sure to capture them for later when you are ready to sketch out your resource payload definitions.

**Relationship Linking**

Relationship links are often useful for clients to explore a resource deeper. While some resources may return related resources in the same response payload, perhaps for mobile client optimization, you should still return links to allow clients to interact directly with them if you offer direct access via an API.

Resources should include links to a parent resource, children, or related resources as identified during our previous taxonomy step (above). These links will allow clients to discover and follow the
Additional Navigation Links

Certain payloads may require additional links beyond those we have already described. This is most common for collection responses, where we may need to offer pagination links for next and previous pages, and for navigation to other areas of the API (e.g. custom resource actions, top-level APIs, the sitemap, etc).

As you design your API, look for opportunities to provide developers with greater insight into your API’s capabilities. Initially, your API may be simple and may not offer many links other than the most obvious ones. However, as your API matures you will find that links will provide developers with a greater insight into new or previously unknown capabilities. This will encourage greater API exploration and integration that will reduce the risk of developers moving to a new and shiny API by providing continual interest and excitement. We cover the patterns around linking and embedding resource relationships in more detail in the upcoming Appendix A, “API Design Patterns”.

Common Questions About API Design

Now, let’s tackle some common questions that we hear during our training and consulting engagements regarding API Design:

Is a great API design really necessary?

Well, yes, you can build and launch your API without spending time modeling and designing it first. In fact, there are several popular APIs that were done in just this way. They must know something we don’t, right? Perhaps. The difference is that companies are now faced with more competition from other API vendors than ever before. A great API design creates a great developer experience, which is a strategic advantage over other API vendors. We recommend spending some time designing your API and applying common patterns and practices so that it is both easy and intuitive.

That said, remember: A great API design that produces no value will never be used. Ensure that your APIs provide value that not only meets expectations but exceeds them.

How much time should I spend designing my API?

There is no specific rule for how long it should take. In fact, the more complex or critical your API, the more time you may need. However, the more times you design APIs the more proficient you will become and the faster the process will progress. We recommend applying these API design guidelines to a variety of small API projects - either real or contrived - to improve your skills. If you take the process all the way to the prototyping stage, you will begin to see the results of your design choices. This will provide more confidence as the API projects become larger or more critical to business success.
How do I handle API Versioning during the API design process?

Your API will likely change over time, as you will never get your API 100% right the first time. Often, you will find that you need to make significant changes to your resource representations, requiring a new version of your API. Versioning of RESTful APIs is a big topic that has started many heated discussions. While there are a variety of ways to version your API, we cover our recommended methods in the appendix, “API Design Patterns” along with the reasons why we choose them. For now, since you are still designing and prototyping, you have much less time and effort invested in these structures. If you need to change - which is quite likely - now is a great time to do it.

Next steps

So far, we’ve discussed how to move from API modeling to design. During this process, we were able to identify the resources we plan to offer in our API, find the relationships between them, and determine the HTTP verbs and response codes that we’ll likely need. All that is left now is to get into the details of our API. That will be the subject of the next chapter, where we start to move from design to prototyping our implementation.
Chapter 7 - Data Architectures

Hypermedia Specifications

Let’s put it out there right now:

**There is no One True Way for hypermedia specifications.**

For every article saying how great one format is, you will find another saying a different format is better. Neither is wrong. The hidden aspect in these conversations are the author’s constraints. Without knowing their requirements, their thought process, and their level of experience and understanding, you don’t know what drove their decisions: it could have been a deep and rigorous understanding of the available options, the supporting tools and documentation, or a compelling presentation at a conference. And remember, when that API company you admire chooses a specification, their decision may have been driven by the same considerations.

Here are the most prominent specifications in alphabetical order.

**Collection+JSON**

Collection+JSON³⁶ was initially created in mid-2011 by Mike Amundsen for the express purpose of serving as a format to support CRUD operations of simple collections. As a result, it tends to focus on four main aspects: collections, templates, queries, and links. A simple Collection+JSON example is available here³⁷.

First and foremost is the **Collection**. A collection is simply a list of resources including details such as their individual urls and fields. This list gives the client application direct access to any item and the necessary information to edit or delete it.

Next are the **Templates** which are the fields to create a valid resource within the collection. With this information, client developers can embed a “new item” form or even create an item request and have confidence that it is correct and the request will be successful.

Next the **Queries** demonstrate how to retrieve and access the collections. A common scenario is providing the mechanism - both the required fields and the url - to create a proper search query. This gives the client consuming the payload a way to quickly perform a search without further analysis or even exploration of the API.

**Links** are the single most fundamental aspect of hypermedia. In the simplest terms, they work identically to links on a website. Clients can use them to delve deeper into the API, determine what options and functionality are available, and potentially how to interact with those end points. Unlike HTML, these links are described with a vocabulary called link relations³⁸. There are a number of

³⁶http://amundsen.com/media-types/collection/
³⁷http://amundsen.com/media-types/collection/examples/
³⁸https://en.wikipedia.org/wiki/Link Relation
link relations - first, last, next, previous - which are formally accepted and defined by IANA\(^39\) but organizations can add their own as necessary.

Like a few of the other specifications, Collection+JSON has been registered through IANA\(^40\) to reserve the media type. While that doesn’t establish any of them as “better” specifications, it does imply a level of rigor and thought has gone into the process.

**JSON for Linked Data (JSON-LD)**

JSON-LD\(^41\) is being actively developed by the W3C JSON-LD Community Group\(^42\) and became a formal W3C Recommendation in January 2014. It was designed to annotate existing JSON documents to make more easily understood and explorable. You can explore a number of examples here\(^43\).

The key aspect of JSON-LD is **Context**. Unlike the other hypermedia specifications, it does not require you to completely rethink and rebuild your documents. Instead, this specification is used to add descriptive information for your existing fields. In the simplest case, if you’ve used schema.org\(^44\) to define your field names and structures, you can simply embed references to the relevant schemas. Alternatively, if you have used your own names, you can add a `@context` key to your payloads and define which fields are what datatypes. While this approach is significantly more effort, you get the same result.

In either case, these definitions give client applications insight into how to display, process, and even validate the fields. For example, if your JSON document includes a `latitude` field, you can reference the `geo`\(^45\) or `latitude`\(^46\) schemas to ensure all client applications share the same understanding with other APIs they may already know. To be clear, JSON-LD does not explain how to use endpoints, it just describes the contents of those endpoints.

Unlike the other specifications, JSON-LD does not have an explicit **Links** section. It is not that links are suddenly unimportant - *it is still hypermedia* - instead it assumes there is already a links section in your payload. In that case, JSON-LD would be used to describe them instead.

**Hypertext Application Language (HAL)**

HAL\(^47\) was created in mid-2011 by Mike Kelly and has been submitted for review\(^48\) to the IETF Network Working Group. Its purpose is to be a lightweight specification that is easy to understand and use. It tends to be characterized as being “HTML for machines.” In terms of implementation details, it revolves around links, embeds, and curies. You can see HAL in production use\(^49\) in a

---

\(^{39}\)http://www.iana.org/assignments/link-relations/link-relations.xhtml
\(^{40}\)http://www.iana.org/form/media-types
\(^{41}\)http://json-ld.org/
\(^{42}\)https://www.w3.org/community/json-ld/
\(^{43}\)http://json-ld.org/playground/
\(^{44}\)http://schema.org/
\(^{45}\)http://schema.org/geo
\(^{46}\)http://schema.org/latitude
\(^{47}\)http://stateless.co/hal_specification.html
\(^{48}\)https://tools.ietf.org/html/draft-kelly-json-hal-07
\(^{49}\)https://github.com/mikekelly/hal_specification/wiki/APIs
number of APIs.

**Disclosure:** HAL is the hypermedia specification that Keith uses in his day job at Clarify.io⁵⁰ so he is probably biased.

The first section is the Curies. They serve as an easy way to include references to documentation from within the API itself. There are no requirements on how that documentation should be retrieved, structured, or presented, but curies make sure we have something available.

The next section includes Embeds. Embeds are simply other resources included directly in the payload of the current document. For example, consider a JSON representation of a book resource. The embed would be individual resources for chapters or pages within the book. As a result, the client application will make fewer requests and potentially speed the overall interaction.

Finally, we have the most important aspect: Links. Links in HAL work identically to links in Collection+JSON with no significant differences. The best part of links is still the ability to explore an API entirely within the API without requiring additional information or external documentation.

### Siren

The Siren⁵¹ specification was started in mid-2012 by Kevin Swiber. At first glance, Siren appears to be the most embryonic of all the hypermedia formats. It does not have an independent website outside of Github, it doesn’t have the same tooling, and it doesn’t have the same adoption. That said, its unique approach makes it valuable and an interesting addition to the toolbox. Like the other specifications it has entities and links but the real power comes from the actions. You can explore an example here⁵².

First are the Entities which are simply a representation of the resources. This includes the fields or properties, the class of resource being shared, and links to additional related resources.

The next section is Links which are identical to links in any of the other specifications.

The most powerful aspect of Siren is its Actions. Actions serve as detailed roadmap for clients to describe and make use of the individual endpoints. While other specifications define required parameters, Siren goes a step further in defining the endpoint, the HTTP verb required, and even a unique name for the form a client application might create. It’s similar to the Templates from Collection+JSON but the additional information lets clients embed forms, take action, and explore and interact with the API to an unprecedented level.

### Considerations and Criteria

When evaluating hypermedia specifications, the single most important aspect is to **keep your requirements in mind**. Do not simply chase a new shiny. Remember there is no “best” specification, just the best one given your use case, understanding, and audience. Given that, here are some considerations to keep in mind:

1. **Are you building a new API or updating an existing one?**

⁵⁰http://clarify.io/
⁵¹https://github.com/kevinswiber/siren
⁵²https://github.com/kevinswiber/siren#example
- If you already have an API, JSON-LD is ideal for annotating existing payloads in a backwards and forwards-compatible way.
- If you are starting completely new, you have a free hand to experiment and explore.

2. What is the primary use case?
   - If your API is read-only - such as Marvel Comics\textsuperscript{53} or Best Buy\textsuperscript{54} - then an entity-centric specification such as Collection+JSON or JSON-LD may be the best option.
   - If the API needs to support standard CRUD operations, Siren or HAL will give client developers deeper understanding.

3. Which specification are you familiar with?
   - If your team is already familiar with a specification, default to that unless you have a compelling reason to do otherwise. New projects are inherently risky so any chance you have to reduce risk, do so.

4. Who is your target audience?
   - If your client developers are internal, you can determine upgrade schedules and drive adoption across the organization.
   - If your client developers are external, using a more established specification can help inspire confidence and provide a level of familiarity and understanding which can drive adoption in the larger community.

5. What tools does your target audience require?
   - If your client developers expect to find language-specific libraries for your API, more common specifications like HAL or Collection+JSON may be the best option.
   - If your client developers are fluent in HTTP or prefer generic HTTP libraries, the specification will be less important in general.

Once again, while there is no right answer to choosing a hypermedia specification, though there is an obvious wrong answer: writing your own. At this stage, there is no reason to create your own specification. The formats described above are based on planning, testing, and usage by some of the sharpest minds dealing with real world constraints and requirements in production APIs. Your team may be great, your understanding of the technology may be superb, and your use case incredibly unique but it is still not a good reason.

Remember your goal: Help people to be successful with your API.

When someone is evaluating or using an API for the first time, they have to learn numerous aspects: authentication, end points, parameters, error handling, and where it fits into their application. If you can simplify the process using familiar patterns, structures, and naming conventions that are already supported by tooling, you accelerate adoption and make success more likely. Remember that their primary goals are to solve their problems and go home.

Selecting a Data Storage Strategy

At the heart of every API is often the need for one or more data stores. While there are a variety of options now available for storing, indexing, searching, and analyzing your data, not every option

\textsuperscript{53}http://developer.marvel.com/
\textsuperscript{54}https://developer.bestbuy.com/
will be right for your API implementation. Let’s look at the variety of data storage options available and how they fit into our overall API architecture.

**Revisiting Relational Databases and SQL**

Until a few years ago, relational databases were the default decision for any software product or IT department. There are a variety of vendor choices from open source databases such as PostgreSQL and MySQL to the more traditional closed source options such as Microsoft SQL Server and Oracle.

Relational databases group data into tables. Tables are composed of columns, which must be typed and can be chosen from a variety of types, from fixed-length strings to variable strings, dates, timestamps, and integer or floating point numbers with a variety of precision options. Tables may be associated to other tables by making the primary key of one table related to a column in another table, often called a foreign key. These relationships are the heart of a relational database and encourage data to be organized around the cardinality of the relationship: one-to-one, one-to-many, and many-to-many. Changes to data in a relational database are grouped into transactions, ensuring that data is either committed or rolled back.

The transactional capability of relational databases is a strong advantage for data integrity, but often results in a strong disadvantage regarding scaling options. Techniques such as clustering, replication, and partitioning are often used to improve the scalability of relational databases.

As the growth of web and mobile applications continued, many relational databases began to require more complex methods of scaling due to the sheer volume of data being inserted and queried per second. This caused a new movement, often called NoSQL, to emerge to try and overcome some of the more common scaling limitations around relational databases.

**Understanding NoSQL**

Non-relational data stores are not a new thing. Directory servers, Object Databases, and XML Databases are three popular alternatives prior to the inception of the NoSQL movement in 2009. Each of these data stores offered different styles of capabilities, storage, interfaces, and performance alternatives to SQL-based data stores.

As websites such as Amazon, Facebook, Google, Digg, and Twitter struggled to handle the increased load of their user base, scalable data storage became more important and eventually trickled down to companies and applications of all sizes. While many SQL database vendors offer clustering and high availability solutions, NoSQL vendors have focused their optimizations by taking a different approach than most relational databases.

Simply defined, NoSQL is a classification of data stores that do not require relational or transactional data storage, usually do not require fixed schemas, and often scale horizontally for greater application throughput and scalability.

While some NoSQL data stores do provide a hybrid of traditional SQL-based database features, many exclude these features in an effort to provide simplicity of data storage and retrieval (and therefore better performance and scalability).

Today, there are a variety of commercial and open source NoSQL data stores that fall into the following categories:
- **Document Stores** - manage collections of documents, often structured as XML, JSON, or other data formats. Examples include: MongoDB, CouchDB, Solr, and Elasticsearch
- **Column-Based Stores** - manage a tuple consisting of a unique column name, value, and timestamp of the data. Examples include: Cassandra and HBase
- **Key-Value Stores** - manage values that are mapped and accessed using a unique key within a collection. Examples include: Amazon DynamoDB, Redis, Riak, and Memcached
- **Graph-Based Stores** - manage graph structures of nodes that are connected using edges and may have associated properties. Examples include: Neo4j, Allegro, and OrientDB

Each of these types of databases provide distinct tradeoffs of how the data will be stored, retrieved, and processed. While relational databases often enable ad hoc queries that may be optimized through the use of indexed columns, not all NoSQL databases offer this capability. Before you select a NoSQL database, be sure to fully analyze and understand the ways in which you will want to access your data from your API. Otherwise, you may select a NoSQL database that will perform slower than a relational or alternative NoSQL-based data store.

### Choosing Between SQL and NoSQL

Many ask if relational databases are no longer necessary or relevant. This likely depends on the type of API you are building. We have found that most applications still require support for ad hoc queries often not available or easily supported by NoSQL databases. In addition, API-adjacent services such as reporting, analytics, and decision support systems are often designed to work only with SQL-based databases.

NoSQL solutions have been primarily focused on storage and less on integrating or offering alternatives to these SQL-based tools (though this is starting to change). Some relational database vendors are finding ways to blend NoSQL concepts, such as JSON-based document and key-value store support, into their existing offerings to provide a hybrid solution that meets the need of modern web and mobile applications. Combined with the fact that SQL is often well known and accepted in our industry, you won’t want to count relational databases out entirely. However, some NoSQL vendors are starting to add limited SQL support on top of their tools, enabling them to expand their reach.

As a result of these trends, we recommend the following approach when selecting a data storage strategy:

1. Analyze your data storage needs to fully understand what data you need to store, how it will be accessed, and other data access requirements
2. Build a candidate list of categories and vendors that may satisfy your API storage requirements
3. Prototype the data storage and common queries, including both simple and complex queries required to implement your API
4. Obtain or build tools or scripts that load test the prototype database with a small number of records initially to test your queries
5. Load the prototype database with a large number of records that will likely represent your needs over the next 12-18 months, benchmarking the queries to identify issues or concerns
Using this approach, we have often identified issues related to data storage, access, and scale earlier in the development cycle. This has saved us several times from being forced to change the database post-production (and migrating production data in the process). The scope of your evaluation should take into account proper data modeling, database setup and configuration, and driver selection for your specific programming language/framework. If your list of candidate databases exceeds 3, we recommend you scope the effort initially to just a few days to narrow the list, then perform a more comprehensive selection process after that.
Chapter 8 - Documenting Your API

From the getting started guide to core concepts and example code, your API documentation informs the audience about how you see, understand, and approach their problem. As we know, we can solve problems using a variety of strategies and points of view. The words and code you choose are important. Your documentation tells me your story: how you see the world, my problems, and your solution to those problems.

The Hidden Value of API Documentation

API documentation is the primary communication medium between API provider and consumer. With the exception of open source APIs, you will likely never see the source code behind it or the assumptions and decisions that went into it. Therefore, the documentation is the primary thing that expresses your priorities, purpose, and intent to the developers consuming your API. Without clear, accurate, and complete documentation, developers will struggle to use your API or even discern its goals.

Documentation also serves as a key step in validating API design before a single line of code is written. By including documentation as part of the process and including technical writers during the design and development process, API design flaws can be spotted before your team has spent time building full systems. It’s easier and less painful to delete a few pages of a document than trash weeks or months of planning and development.

API Definition Formats

While you could capture your API documentation in a static format such as documents or spreadsheets, better API documentation tools now exist. There are API definition formats to capture your thought process and decisions and later generate reference documentation. Further, many of these tools offer an option for providing interactive documentation to allow both developers and non-developers to explore your API. All of this before writing a single line of code.

As an additional incentive, the API definitions also offer the capability to drive automation tools and server code generation. Code generation may be used to generate server stubs for your favorite API framework, client code for static SDKs, or even the emerging trend of dynamic SDKs that are generated “on-the-fly” from the definition. In most cases, these are not “one and done” tools that generate code at the beginning of your project and quickly set aside. These tools can be used with the definition formats throughout the life of the project to keep the team, system, and supporting documentation synchronized with little effort.

As of this writing, there are several popular API definition formats:

1. Open API Initiative⁵⁵ - formerly known as Swagger

⁵⁵https://openapis.org/
Each of these formats, as well as lesser-known formats, offer different tradeoffs with regard to API documentation. Some focus simply on the generation of the documentation itself using a simple format, while others focus on a more strict structure enable the code generation tooling. Like in many other situations, there is no “right answer” but instead an answer that will make sense for your team building your project at the time.

**Documenting Your API Design**

When documenting your API during the design process, the focus should be to take a breadth-first approach by first identifying and documenting all of the available endpoints. This will ensure that you are able to see your API design emerge quickly and spot any trouble spots early.

Start by documenting the endpoints for each resource, the HTTP verbs that will be supported as part of the lifecycle, and all relevant success and error response codes. This should be straightforward, as much of this has already been put together from the API design exercise in Chapter 6.

After this stage, you may wish to start filling in all of the details about expected parameters and representation formats. As your API documentation begins to emerge, you should share it with other teams for review. Now is the cheapest and easiest time to find and fix errors, incorrect assumptions, and misunderstandings. You can only accomplish this if the developers, QA, project managers, scrum masters, and product owners have the opportunity and motivation to review the documentation and provide feedback for improvement. You should also share your documentation with other teams within the organization and trusted external developers that can provide important feedback on your design and overall documentation approach. Small corrections at this stage can save on later development and accelerate adoption.

**Selecting an API Definition Format**

While we have used a variety of definition formats, we find selecting an API definition format is about finding the best fit for your team and the specific API you are designing and building. To make a good decision, it requires some important considerations. We recommend applying the following criteria to help with your selection process:

1. The format should have clear guidelines and documentation on how to use it, common patterns and practices, and an active community building tools to ease adoption.
2. The format should encourage continued use. Just because the format is popular right now doesn’t mean your team will stick with using it over time. The format should encourage all team members - not just developers - to keep the documentation up-to-date to avoid incorrect or incomplete docs.

56[^56]
57[^57]
58[^58]

[^56]: http://raml.org/
[^57]: http://apiblueprint.org
[^58]: http://www.mashery.com/product/io-docs
3. The tooling should be based on technologies that your team is comfortable installing, using, and perhaps customizing or extending over time.

4. Rendered documentation should be clear, easy-to-navigate, and allow for interactive exploration of your API.

5. For documenting existing APIs, determine if tools exist for reverse-engineering your API usage into the format to ease getting started and ongoing maintenance.

Your team should evaluate one or more formats to reach a decision that will help the team now and into the future. If your primary concern is the initial design process, you are ignoring the needs and issued around development and ongoing maintenance, which is the vast majority of the lifetime of your API. Keep in mind that while each format is different, many of them are close enough to one another that a simple script or service can be used to convert one format to another should you need to switch.

**Beyond API Reference Documentation**

We use the term “API documentation” as if there is only one kind of documentation. Yes, you need to deliver a great API reference for developers. However, complete API documentation requires more than just a API reference in HTML or PDF form. As Keith describes in “Every API has Three User Interfaces⁵⁹,” there are three primary times when someone needs your documentation:

**Evaluation** - When a product manager or developer is attempting to determine if your API solves their problem. If you cannot clearly and concisely explain the value of your solution, they are lost as a customer.

**Integration** - Once they understand that your API solves their problem, they’ll want to try solving their problem. This will start as simple experimentation and - if successful - proceed to a full integration.

**Debugging** - Once your API is embedded into their application, the next time they’ll will visit your documentation is when something goes wrong. It can be as simple as unexpected error messages to a major issue or outage.

To accomplish all of these successfully, teams must understand that documentation isn’t just a “dump” of API parameters. There are marketing and customers support aspects that can be used to help throughout the customers’ journey:

**Features and Discovery** – Provides an overview of the API, addressing concerns such as benefits, capabilities, and pricing of your API to qualify prospects.

**Case Studies** – Case studies highlight successful applications that have been built using your API.

**Reference Docs** – Provides a reference for each API endpoint to developers, including details on the URL, HTTP verb(s) supported, response codes, and payload formats.

**Examples and Guides** – As a user new to your API, the most difficult part is the initial learning curve. Guides offer help with learning an API’s concepts and vocabulary during this critical stage.

⁵⁹[http://clarify.io/blog/every-api-company-has-three-user-interfaces/]
**Problem/Resolution** – Sometimes developers need a little help when consuming your API. Documentation that helps developers troubleshoot common issues and explain error response codes can ease the burden on your developers and support staff.

**Internal Docs** – For those instances when developers need more assistance with your API, it is important to have internal assets to help your support team.

You can learn about each of these in more detail from James’ blog post titled *Moving Beyond API Reference Documentation*[^60]. In addition, the post titled “*10 Questions Your API Must Answer*”[^61] provides a guide to assessing the quality of your API documentation across your website and developer portal.

[^61]: http://launchany.com/10-questions-your-api-document-must-answer/
Chapter 9 - Prototyping Your API

As your design starts to emerge, the temptation is to move directly into development. Don’t. A proper and complete API design is critical to team and customer communication. If your team rushes into development too soon and without proper feedback, valuable time and resources will be wasted. Or worse, your team will build and release something that has to be maintained and supported for months or years. There are many companies held back by poor API choices five or ten years ago. You don’t want to join that club.

API design provides an opportunity for teams to communicate and adjust the design prior to the implementation effort. At this stage, changes can be made by editing words on a page instead of destroying weeks or months of work. We recommend taking the time to prototype your API before you begin development, as it provides several advantages:

1. Communicates the API deliverables to API consumers early, allowing for internal and external feedback before development begins.
2. Improves the developer experience by encouraging an outside-in perspective on your design, separate from internal implementation details.
3. Identifies missing API endpoints early rather than later in the development process.
4. Prioritizes development sprints based on external dependencies to specific endpoints.
5. Parallelizes efforts outside of development, enabling more efficient use of testing, documentation, and marketing teams.
6. Parallelizes efforts within development allowing teams building supporting components such as helper libraries, web UIs, etc.

There are three primary approaches to prototyping an API:

1. API Documentation
2. Building small, static prototypes
3. Building a partial or full working prototype

We generally recommend that you use two of the three prototyping methods for any non-trivial API. We’ve already covered the API documentation approach (covered in the previous chapter) so let’s look at the static and working prototyping methods and how they can result in a better API design.

Static Prototyping

Once the overall API endpoint design has been documented, shared, and feedback incorporated, the next step in the prototyping process is to generate static prototypes.
Static prototyping allows us to focus on the overall representation format of our resources. This technique is useful to help explore and build the contract between your API and mobile or web applications that will consume them. It is also useful for creating mock responses to portions of our API for automated test coverage.

When building static prototypes of your resources, be sure to create both a sample collection view, as well as a few individual resource representations. This will provide the most flexibility and help you spot the areas where you may wish to reduce resource representations returned in a collection, requiring clients to retrieve the full instance representation.

That said, since we cannot create, update, or delete our resources using a static prototype, this technique does have limited usage. We recommend using static prototypes to work out the resource representation details, then moving to a working prototype to finalize the overall API design and provide full lifecycle support for early web and mobile integration. Often, the static prototypes can be used to help develop the working prototype.

At this stage, it's worth remembering the constraints of our intended clients. If the API is for mobile devices, you should not assume a high speed or even a reliable connection. As you design your payloads, this may lend you to embedding sub resources either automatically or upon request. Keep this in mind as you structure both the payloads and the requests.

**Methods For Static Prototyping**

Since static prototyping only requires serving a file with resource definitions, there are a variety of options for serving up these static files:

1. A local or shared file system
2. A local or remote web server
3. Hosted object storage using Amazon S3 or equivalent

In addition, we have had some success by using Heroku’s Data Clips\(^6^2\) feature to serve JSON files from data stored in a hosted PostgreSQL database. Google Spreadsheet’s JSON feed support\(^6^3\) may be another option to allow group collaboration and maintenance of the data while feeding JSON to an application. However, these approaches often have limitations on the amount of control available when crafting the resource representation.

**Building a Working Prototype**

As you prototype, you will want to spend time focusing on no more than one or two of the following areas:

1. The resources that your API will offer, including nested summaries for related resources
2. Resource payloads containing the most important state representation

---

\(^6^2\)https://blog.heroku.com/archives/2012/2/14/simple_data_sharing_with_data_clips
\(^6^3\)https://developers.google.com/gdata/samples/spreadsheet_sample
3. Hypermedia links that will drive client behavior and access to other resources
4. Response codes for success and error cases
5. Acceptance tests that implement use cases that the final API must deliver

By choosing specific areas you want to focus on, you will prevent the prototype from becoming a full implementation and instead focus on what is most critical: feedback on your API design prior to implementation.

Selecting a Prototyping Framework

When building a working prototype, it may be tempting to use the existing libraries and frameworks that will be used to produce the production API. For teams that are using a more complex, inhouse library that is meant for a production environment, this will slow down the prototyping process. The habits and practices that your team has and that your frameworks encourage will drive towards doing things the “right way” instead of working to explore and understanding. In these instances, we recommend reducing the prototype code to the minimum required to obtain feedback quickly.

Think of yourself as MacGyver trying to test and figure out a solution to one particular problem. And once you know the answer, you must be willing to replace it with a more formal and complete solution later.

Consider using only the minimum tools and frameworks required to prototype the API. In some cases, you may want to experiment with a simple framework in a language you don’t often use just to discourage the idea that this is a production-ready application.

Specifically, consider using languages and tools such as Sinatra in Ruby, hapi.js for Node.js, or Slim or Silex for PHP to generate simple, quick APIs. This will help to accelerate the communication loop and prevent details and constraints of the final implementation from slowing the team down. Remember, your API and client applications speak HTTP, which is programming language agnostic. Take advantage of this fact by looking outside your day-to-day tools to accelerate team learning.

To provide some assistance in getting started with your prototype, you may wish to consider Kirsten Hunter’s Polyglot Github repository⁶⁴ as a starting point. This repository was built to complement her talk that encourages developers to learn more than one programming language. She implemented the same API across PHP, Python, Perl, Ruby, and Node.js. It is backed using MongoDB, a document-based database well suited for prototyping APIs. It serves as an easy way to explore and review common issues in a completely platform-agnostic approach.

⁶⁴https://github.com/synedra/polyglot/
Chapter 10 - API Product Management

As we wrap-up, we want to address the team dynamics related to the API design process. Fundamentally, the goal is to help teams benefit from the communication, shared understanding, and opportunity to catch and prevent errors that API design affords.

The API-First Organization

As we have mentioned previously, an API-first architecture is an approach that treats the API as a goal and a product offering of its own. It is not a bolt-on-after-the-fact checkbox that is dealt with at the end of the project. An API-first approach focuses on a well-designed and well-documented API that offers a great developer experience (DX) for developers building web or mobile applications. These APIs serve as the internal or external developers including partners and customers.

An API-first organization sees APIs not just as an important part of an existing product, but rather first-class products themselves. This means surrounding the API with a full product strategy, governance, and support structure that includes many of the following: program managers, usability experts, technical support staff, and a sales/marketing team with developer evangelists and sales engineers. By all definitions, it is a product and should be treated as such.

For early stage companies, the decision to become an API-first organization may be more obvious and likely an easier transition as the team is smaller. For companies that have existing products or IT departments, going API-first doesn’t happen overnight. The transition often requires identification of candidate APIs, training existing staff in API and API-first techniques, and building teams to design, build, and support new internal or public API offerings. The remainder of this chapter focuses on common challenges and techniques found within API-first organizations. However, we recommend seeking out assistance in transitioning to an API-first organization to accelerate learning and reduce the likelihood of API launch failures.

For background, we developed and learned these lessons over multiple years training thousands of people across North America. Those people included over a dozen different job roles/titles and James has detailed some of these lessons in depth on his blog⁶⁵.

Front-End, Back-End, and Full Stack Developers

It seems obvious but it’s worth saying: different developers approach different parts of the system at different times.

Front-end developers spend their time where design meets development. These developers focus on HTML, CSS, and Javascript - basically everything a user sees and interacts with - which means they often consume rather than produce APIs. While they may write server-side code, it is often

restricted to the server-side controllers that act as the intermediary between the front-end and back-
end. They prefer to talk about design, user experience, and using frameworks such as Backbone.js, 
Angular.js, etc to build applications. Some front-end developers also work in one or more mobile 
development roles as well. This role is becoming more important as the need for well-designed 
applications continues to grow. As a result, they have certain needs and use cases in mind for the 
API that are required to complete or simplify their own development.

On the opposite spectrum are the back-end developers who focus on databases, cloud/server 
infrastructure, and system integration. While some may appreciate an application with great design, 
they tend to stay away from anymore more than rudimentary work with HTML, CSS, and Javascript. 
Back-end developers often work with a variety of programming languages, such as Java, Python, 
Ruby, GoLang, Scala, Node.js (server-side Javascript), and others. They also work with one or more 
database vendors, both SQL and NoSQL-based. When they think about API design and usage, it’s 
generally centered around communicating with other systems and exchanging data, far away from 
user interfaces and the related concerns.

Between these two extremes is the full-stack developer. This type of developer has a view of 
the system that spans the entire system. They are generally proficient in HTML, CSS, Javascript, 
and then at least one backend language. They understand and can work across both disciplines 
of application development. While they may not be specialists, or only specialists in one area, they 
understand both front-end and back-end technologies and related impact of decisions that are made. 
On rare occasion, a full-stack developer may have specialization in both areas and are often recruited 
by early-stage startups due to their broad range of skills in design and back-end development and 
flexibility to serve in a variety of roles. As teams expand and individuals begin to specialize, front 
and backend developers are often added to fill gaps and get the depth required for longer term 
success.

![Diagram of Front-End, Back-End, and Full Stack Developer]

The Typical Front-End, Back-End, and Full Stack Developer
API-First organizations will benefit from a mixture of all three styles of developer. Front-end developers will focus on the integration of APIs into web and mobile applications and can provide feedback on API design and developer experience (DX). Back-end developers often focus on data storage, real-time and batch data processing, and cloud infrastructure. Full stack developers are flexible and can move between both environments over time, bringing the team together and filling in gaps for existing teams as necessary. A well-balanced team representing each of these needs is a powerful combination that can detect and address concerns early in the process potentially saving massive amounts of time and money throughout the entire product lifecycle.

To be blunt: *A team communicating and working together on Day One will save you hundreds of hours and tens of thousands of dollars at minimum.*

### Partitioning API Teams

As the number of APIs grow within an organization, it becomes more difficult for developers to improve and support every aspect of an API. As a result, organizations often look at partitioning out APIs into functional areas, allowing teams to grow and support specific aspects of an API. This often requires separating the API codebase into isolated systems. The method of division between APIs may be functional, third-party related, or based on data or technology limitations.

As an example, in the early days at Twilio, they separated the API teams into three core areas: inbound SMS, outbound SMS, and Voice. This excludes non-API teams for reporting, billing, and operations, etc. The reason for the separation is that the skills, understanding, and constraints necessary to integrate with SMS network providers are much different than those related to voice, and within SMS the skills vary between inbound and outbound messaging and all of them allow or require different architectures and technologies.

For example, the definition of “real-time” varies in these systems. If you send a text message, you expect quick delivery and five or even fifteen seconds is acceptable. For a voice call, a delay as little as 200 milliseconds makes the call seem wrong as you talk over your friend and vice versa.

### Accelerating API Integration

One of the things that you will discover when building an API is that often the API is on the critical path for many other teams in the organization. Mobile and front-end Javascript developers need to build their UI components; QA can’t execute on their test plans; and the documentation can’t be written until the API is ready. Even marketing and operations can be held up if an API hasn’t been developed yet.

Fortunately, there are a variety of ways around this, many of which we have mentioned throughout the book. For easy reference, the collection of these recommendations to help your team parallelize the full product lifecycle are reiterated below:
Model the API with Your Entire Team

As we discussed in Chapter 4, the process of defining the requirements, scope, and activities that your API provide should include your entire team. Business software analysts, along with subject matter experts, will help provide insight to the full team on what the API should offer to be successful. This collaborative effort will expedite knowledge transfer between team members, address concerns before they become problems, assist in developing early drafts of technical documentation, allow QA to prepare test plans, and speed development effort by reducing misunderstanding and allowing the teams to work in parallel. An open but focused effort on API modeling early in the process is one of the single most valuable things you can do.

Taking a Design-First Approach

Taking the time to refine the API design not only provides a more usable API for developers, but it will decrease API delivery time. API documentation and prototyping techniques, discussed in Chapter 8 and Chapter 9 respectively, allow the API design to be shared early in the process in a standard way with a variety of team members. API definition formats, such as OpenAPI (the format formerly known as Swagger) and RAML, can be used to generate documentation for review for internal and external stakeholders, as well as for the generation of boilerplate client and server code.

Prototyping the API

Prototypes are not only for validating the API design, but also for early integration with mobile and front-end Javascript and mobile developers long before the API is ready. Prototypes often take on two forms: throwaway prototypes and evolutionary prototypes.

Throwaway prototypes are built as a quick way to validate the design but must eventually thrown away once the production API development achieves an acceptable level of completion. In more colloquial terms, this is the MacGyver version of your API. We recommend using a technology stack that is designed for quick iteration and experimentation instead of your normal tool choices. Don’t worry about performance or long term operations, just experiment to see where issues do and don’t arise in the process.

Evolutionary prototypes, however, are built from the beginning to use the target technology stack for the production environment, with the focus of vetting the API design rather than for implementation completeness. Over time, the evolutionary prototype will become more production-ready until no sign of any prototype code remains.

QA can use both kinds of prototypes to begin to develop manual test scripts and automated testing. Operations teams can use evolutionary prototypes, built with production technologies in mind, to work out many of the infrastructure details, configuration scripts, and other automation requirements.
Prototyping Your Way To Implementation

We cannot recommend prototyping your API enough, as we have found it encourages communication and exposes problems early, thereby reducing risk. It also allows you to create rough drafts of helper libraries, sample documentation, example code and other supporting components to the overall product.

As your team beings to design and build new APIs often, you will start to move from modeling to design to documentation quickly, allowing you to move through this process routinely as you design and implement new APIs. This rapid approach will give you more confidence that what you are designing will meet the needs of your internal and external developers.

Test Your APIs Early

Teams often see the API workflow as a serial process: model, design, develop, test, and document. However, with good API design techniques, we can move most of these tasks forward in your project plan to parallelize the overall effort and increase the odds that you ship on time and on budget.

Testing APIs can be divided into three areas, which with a specific focus on what is being tested:

**Unit (code-level) testing:** Tests concerned with the internal functionality of the API. This is commonly called unit testing or controller testing and focuses on specific areas of the code. For APIs, this means testing the API by testing the code that makes up the API. White-box testing should happen during the development process.

**Functional testing:** Tests concerned with the inputs and outputs of the API. This is often called functional testing or integration testing. The tests are limited in scope to a specific endpoint and are conducted over HTTP rather than against the internal code. Black-box testing benefits from a working prototype, as it allows dedicated QA teams to build automated testing in parallel with development.

**Acceptance testing:** Verifies that all business requirements are met. These tests are often scenario-driven, focus on how the end-user or developer will use the API, and include validating that the API will support all web and mobile functionality. As with black-box testing, acceptance testing benefits from working prototypes to parallel QA efforts. These tests can also serve as another form of documentation demonstrating common use cases and the resulting code involved.

**Load testing:** Identifies areas of weakness when the API is under load. Hotspots such as API logic, database design, and infrastructure limitations should be identified and addressed to ensure that your API will perform properly at and beyond expected load.

Just like every other task and process we’ve described, incorporating teams earlier in the process creates massive benefits both short and long term. Any issue your team can detect and prevent now will save you in development, maintenance, and customer support.
Closing.. for now

The document you hold in your digital hands is not quite complete. There was a long delay between the second and third releases and that’s my (Keith’s) fault. It turns out that combining a book with a startup with a new child is a bad combination. Regardless, we have one more section - API Design Patterns - to finish and we’re working diligently on that now.

_Luckily, because we are using LeanPub, you will receive that and any other updates for free._

As a result, we hope this book becomes a conversation as we write. When you find something interesting, let us know. If you think we’re wrong, say so. If you think we’re right, tell your friends. If you want us to teach your team more and deeper concepts, please let us know.

Towards that goal, we have one remaining chapter on our drawing board:

- API Design Patterns
  - Idempotency
  - SOLID Design Principles
  - Naming - Resources & Parameters
  - Transactions
  - Authorization - Tokens vs Credentials
  - Link Relations
  - Composites
  - Versioning
  - Pagination
  - Error Handling
  - Caching - Strategies & ETags
  - Hard vs Soft Deletes
Want to stay informed about everything API-related?

While you wait, why not subscribe to the free API Developer Weekly? It is our hand-curated weekly email that is hyper-focused on the business, design, development, and deployment of APIs for web and mobile apps. It’s great for developers, architects, product managers, and executives and serves as an easy way to stay up to date for free. View a sample newsletter and subscribe.66