
Undisturbed REST
A guide to designing the perfect API
{"by":"Michael Stowe"}
I wish I had read this kind of book before starting my first API project!
- Arnaud Lauret (@apihandyman)

© 2015. All Rights Reserved.
Undisturbed REST
First-look Edition
May 2015, PDF
ISBN 978-1-329-11656-6

To learn more, visit us online at http://www.mulesoft.com/restbook

Published by MuleSoft, 77 Geary Street, Suite 400, San Francisco, CA 94108

A Note from the Author

First and foremost, I want to thank you—the reader—for taking time out of your busy schedule to read this book. Undisturbed REST is not designed to be a tell-all book that covers every line of code in regards to building your API, but rather a book that takes you back to the basics and focuses on how to not just build an API, but build a long-lived API that your users will love. An API that is carefully created to be extendable and flexible, and will save your users time, money and energy in the long run.

As this is the very first edition of this book, I would ask that if you find mistakes you are both willing to forgive them and willing to let me know so that they can be corrected. This feedback will be very much appreciated and can be directed to me@mikestowe.com with the subject "Errata."

Please also understand that while I would love to answer any questions, this may not be feasible from a time standpoint. But I would encourage you all to post freely in the MuleSoft forums (under API), and I, a member of my team or someone from our community will be glad to help out.

I would also like to thank Dr. Roy Fielding for being extremely gracious in letting me include his original doctoral dissertation diagrams in this book to demonstrate how the different constraints of REST work together.

I'd also like to thank Arnaud Lauret (@apihandyman) and Jason Harmon (@jharmn) for taking the time to review this book prior to publication. As well as a very big thank you to Phil Hunter who spent countless hours copy-editing, and Chris Payne (@chrisypayne) who meticulously designed the book's amazing cover.

Last but not least, I want to thank MuleSoft for giving me time to write this book, and for making it freely available. I have always believed in the power of community, and been thankful for all of the valuable lessons I have learned from it over the years. I want to also thank my many mentors, my family, and my friends for their patience, love, and support.

Please enjoy this version of Undisturbed REST, a Guide to Designing the Perfect API—a book that is dedicated both to you and the amazing developer community as a whole.

– Mike Stowe (@mikegstowe)

Table of Contents

1. What is an API **1**
Web APIs 3
REST 6

2. Planning Your API **15**
Questions to Ask 16

3. Designing the Spec **25**
Versioning 26
Spec-Driven Development 30
Choosing a Spec 35

4. Using RAML **39**
Query Parameters 43
Responses 44
ResourceTypes and Traits 45

5. Prototyping and Agile Testing **49**
Getting Developers Involved 53
Getting Feedback 56

6. Authorization and Authentication **59**
Generating tokens 62
OAuth2 64

OAuth and Security 68
Adding OAuth to RAML 71

7. Designing Your Resources 73
Nouns 74
Content-types 76
Versioning 83
Caching 88

8. Designing Your Methods 91
Items vs Collections 92
HTTP Methods 93

9. Handling Responses 101
HTTP Status Codes 102
Errors 105

10. Adding Hypermedia 117
HATEOAS 121
Hypermedia Specs 124
Hypermedia Challenges 136

11. Managing with a Proxy 139
API Access 140
Throttling 142
SLA Tiers 144
Security 147

12. Documenting and Sharing Your API 153
API Console 166
API Notebook 170
Support Communities 173
SDKs and Client Libraries 174

13. A Final Thought 179

Appendix: More API Resources 183

Appendix: Is Your API Ready? 187

1

What is an API?

In the simplest of terms, API is the acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other. In fact, each time you check the weather on your phone, use the Facebook app or send an instant message, you are using an API.

Every time you use one of these applications, the application on your phone is connecting to the Internet and sending data to a server. The server then retrieves that data, interprets it, performs the necessary actions and sends it back to your phone. The application then interprets that data and presents you with the information you wanted in a human, readable format.

What an API really does, however, is provide a layer of security. Because you are making succinct and explicit calls, your phone's data is never fully exposed to the server, and likewise the server is never fully exposed to your phone. Instead, each communicates with small packets of data, sharing only that which is necessary—kind of like you ordering food from a drive-

through window. You tell the server what you would like to eat, they tell you what they need in return and then, in the end, you get your meal.

Many Types of APIs

There are many types of APIs. For example, you may have heard of Java APIs, or interfaces within classes that let objects talk to each other in the Java programming language. Along with program-centric APIs, there are also Web APIs such as the Simple Object Access Protocol (SOAP), Remote Procedure Call (RPC), and perhaps the most popular—at least in name—Representational State Transfer (REST).

While the function of an API may be fairly straightforward and simple, the process of choosing which type to build, understanding why that type of API is best for your application, and then designing it to work effectively has proven to be far more difficult.

One of the greatest challenges of building an API is building one that will last. This is especially true for Web APIs, where you are creating both a contract between you and your users and a programming contract between your server and the client.

In this book, we'll take a look at some of the different types of APIs, but then we'll switch gears and focus on building a REST API as defined by Dr. Roy Fielding. We'll cover important principles that are often ignored—and while some of these may seem a little painful or like they just create more work, you'll find that by adhering to these best practices you will not only create a better API, but save a lot of time and money doing so.

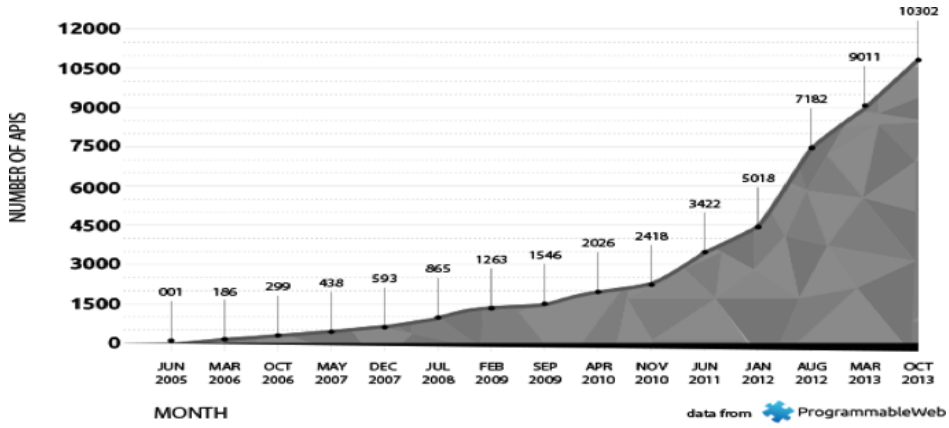
So without further ado, let's get started.

Web APIs

A Web API, otherwise known as a Web Service, provides an interface for Web applications, or applications that need to connect to each other via the Internet to communicate. To date, there are over 13,000 public APIs that can be used to do everything from checking traffic and weather, to updating your social media status and sending Tweets, to even making payments.

In addition to the 13,000 public APIs, there are hundreds of thousands more private Web APIs, or APIs that are not available for consumption by the general public, that are used by companies to extend their capabilities and services across a broad scope of use-cases, including multiple devices. One of the most common forms of a private cross-device Web APIs would be an API written for a mobile application that lets the company transmit data to the app on your phone.

Since 2005, the use of Web APIs has exploded exponentially, and multiple Web formats and standards have been created as technology has advanced:



Early on, one of the most popular enterprise formats for APIs was SOAP. With the emergence of JavaScript Object Notation (JSON), we saw more reliance on HTTP and the growth of JSON-RPC APIs, while REST has grown in popularity and quickly become the de facto standard for general Web APIs today.

SOAP

SOAP was designed back in 1998 by Dave Winer, Don Box, Bob Atkinson and Mohsen Al-Ghosein for Microsoft Corporation. It was designed to offer a new protocol and messaging framework for the communication of applications over the Web. While SOAP can be used across different protocols, it requires a SOAP client to build and receive the different requests, and relies heavily on the Web Service Definition Language (WSDL) and XML:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">

<soap:Body xmlns:m="http://www.example.com/weather">
  <m:GetWeather>
    <m:ZipCode>94108</m:ZipCode>
  </m:GetWeather>
</soap:Body>

</soap:Envelope>
```

Early on, SOAP did not have the strongest support in all languages, and it often became a tedious task for developers to integrate SOAP using the Web Service Definition Language. However, SOAP calls can retain state, something that REST is not designed to do.

XML-RPC

On the other hand, Remote Procedure Calls, or RPC APIs, are much quicker and easier to implement than SOAP. XML-RPC was the basis for SOAP, although many continued to use it in its most generic form, making simple calls over HTTP with the data formatted as XML.

However, like SOAP, RPC calls are tightly coupled and require the user to not only know the procedure name, but often the order of parameters as well. This means that developers would have to spend extensive amounts of time going through documentation to utilize an XML-RPC API, and keeping documentation in sync with the API was of utmost importance, as otherwise a developer's attempts at integrating it would be futile.

JSON-RPC

Introduced in 2002, the JavaScript Object Notation was developed by State Software, Inc. and made most famous by Douglas Crawford. The format was originally designed to take advantage of JavaScript's ability to act as a messaging system between the client and the browser (think AJAX).

JSON was then developed to provide a simple, concise format that could also capture state and data types, allowing for quick deserialization.

Yahoo started taking advantage of JSON in 2005, quickly followed by Google in 2006. Since then JSON has enjoyed rapid adoption and wide language support, becoming the format of choice for most developers.

You can see the simplicity that JSON brought to data formatting as compared to the SOAP/ XML format above:

```
{“zipCode” : “94108”}
```

However, while JSON presented a marked improvement over XML, the downsides of an RPC API still exist with JSON-RPC APIs, including tightly coupled URIs. Just the same, JSON-APIs have been widely adopted and used by companies such as MailChimp, although they are often mislabeled as “RESTful.”

REST

Now the most popular choice for API development, REST or RESTful APIs were designed to take advantage of existing protocols. While REST can be used over nearly any protocol, it typically takes advantage of HTTP when used for Web APIs. This means that developers do not need to install libraries or additional software in order to take advantage of a REST API.

As defined by Dr. Roy Fielding in his 2000 Doctorate Dissertation, REST also provides an incredible layer of flexibility. Since data is not tied to methods and resources, REST has the ability to handle multiple types of calls, return different data formats and even change structurally with the correct implementation of hypermedia.

SOAP	RPC	REST
Requires a SOAP library on the end of the client	Tightly coupled	No library support needed, typically used over HTTP
Not strongly supported by all languages	Can return back any format, although usually tightly coupled to the RPC type (ie JSON-RPC)	Returns data without exposing methods
Exposes operations/ method calls	Requires user to know procedure names	Supports any content-type (XML and JSON used primarily)
Larger packets of data, XML format required	Specific parameters and order	Single resource for multiple actions
All calls sent through POST	Requires a separate URI/ resource for each action/ method.	Typically uses explicit HTTP Action Verbs (CRUD)
Can be stateless or stateful	Typically utilizes just GET/ POST	Documentation can be supplemented with hypermedia
WSDL - Web Service Definitions	Requires extensive documentation	Stateless
Most difficult for developers to use.	Stateless	More difficult for developers to use.
	Easy for developers to get started	

As you can see in this chart, each type of API offers different strengths and weaknesses. REST, however, provides a substantial amount of freedom and flexibility, letting you build an API that meets your needs while also meeting the needs of very diverse customers.

Unlike SOAP, REST is not constrained to XML, but instead can return XML, JSON, YAML or any other format depending on what the client requests. And unlike RPC, users aren't required to know procedure names or specific parameters in a specific order.

But you also lose the ability to maintain state in REST, such as within sessions, and it can be more difficult for newer developers to use. It's also important to understand what makes a REST API RESTful, and why these constraints exist before building your API. After all, if you do not understand why something is designed in the manner it is, you are more likely to disregard certain elements and veer off course, often times hindering your efforts without even realizing it.

Understanding REST

Surprisingly, while most APIs claim to be RESTful, they fall short of the requirements and constraints asserted by Dr. Roy Fielding. One of the most commonly missed constraints of REST is the utilization of hypermedia as the engine of application state, or HATEOAS, but we'll talk about that in another chapter.

There are six key constraints to REST that you should be aware of when deciding whether or not this is the right API type for you.

Client-Server

The client-server constraint operates on the concept that the client and the server should be separate from each other and allowed to evolve individually. In other words, I should be able to make changes to my mobile application without impacting either the data structure or the database design on the server. At the same time, I should be able to modify the database or make changes to my server application without impacting the mobile client. This creates a separation of concerns, letting each application grow and scale independently of the other and allowing your organization to grow quickly and efficiently.

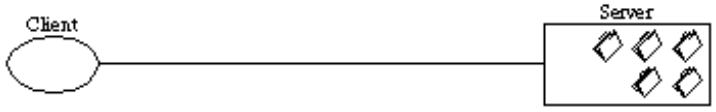


Figure 5-2. Client-Server

Stateless

REST APIs are stateless, meaning that calls can be made independently of one another, and each call contains all of the data necessary to complete itself successfully. A REST API should not rely on data being stored on the server or sessions to determine what to do with a call, but rather solely rely on the data that is provided in that call itself.

This can be confusing, especially when you hear about using hypermedia as the state of the application (Wait—I thought REST was stateless?)..Don't worry, we'll talk about this later, but the important takeaway here is that sessions or identifying information is not being stored on the server when making calls. Instead, each call has the necessary data, such as the API key, access token, user ID, etc. This also helps increase the API's reliability

by having all of the data necessary to make the call, instead of relying on a series of calls with server state to create an object, which may result in partial fails. Instead, in order to reduce memory requirements and keep your application as scalable as possible, a RESTful API requires that any state is stored on the client—not on the server.

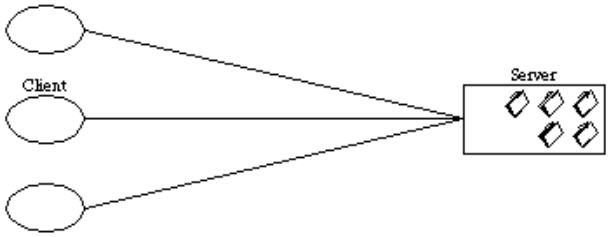


Figure 5-3. Client-Stateless-Server

Cache

Because a stateless API can increase request overhead by handling large loads of incoming and outbound calls, a REST API should be designed to encourage the storage of cacheable data. This means that when data is cacheable, the response should indicate that the data can be stored up to a certain time (expires-at), or in cases where data needs to be real-time, that the response should not be cached by the client.

By enabling this critical constraint, you will not only greatly reduce the number of interactions with your API, reducing internal server usage, but also provide your API users with the tools necessary to provide the fastest and most efficient apps possible.

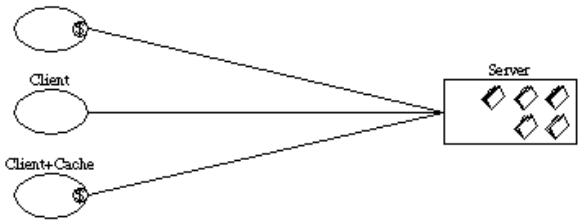


Figure 5-4. Client-Cache-Stateless-Server

Keep in mind that caching is done on the client side. While you may be able to cache some data within your architecture to perform overall performance, the intent is to instruct the client on how it should proceed and whether or not the client can store the data temporarily.

Uniform Interface

The key to the decoupling client from server is having a uniform interface that allows independent evolution of the application without having the application's services, or models and actions, tightly coupled to the API layer itself. The uniform interface lets the client talk to the server in a single language, independent of the architectural backend of either. This interface should provide an unchanging, standardized means of communicating between the client and the server, such as using HTTP with URI resources, CRUD (Create, Read, Update, Delete) and JSON.

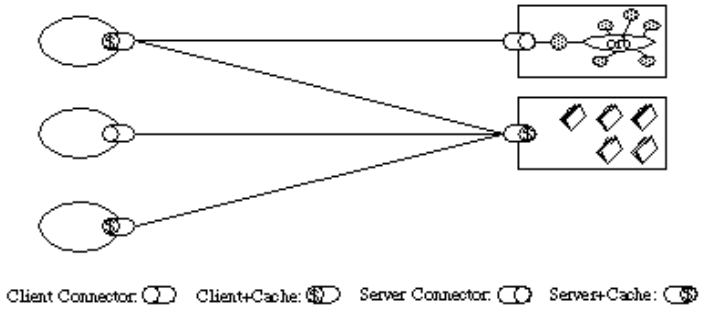


Figure 5-6. Uniform-Client-Cache-Stateless-Server

However, to provide a truly uniform interface, Fielding identified four additional interface constraints: identifying resources, manipulation through representations, self-describing messages and HATEOAS (hypermedia as the engine of application state).

We'll talk about these more in a later chapter.

Layered System

As the name implies, a layered system is a system comprised of layers, with each layer having a specific functionality and responsibility. If we think of a Model View Controller framework, each layer has its own responsibilities, with the models comprising how the data should be formed, the controller focusing on the incoming actions and the view focusing on the output. Each layer is separate but also interacts with the other.

In REST, the same principle holds true, with different layers of the architecture working together to build a hierarchy that helps create a more scalable and modular application. For example, a layered system allows for load balancing and routing (preferably through the use of an API Management Proxy tool which we'll talk about in Chapter 11). A layered system also lets you encapsulate legacy systems and move less commonly accessed functionality to a shared intermediary while also shielding more modern and commonly used components from them. A layered system also gives you the freedom to move systems in and out of your architecture as technologies and services evolve, increasing flexibility and longevity as long as you keep the different modules as loosely coupled as possible.

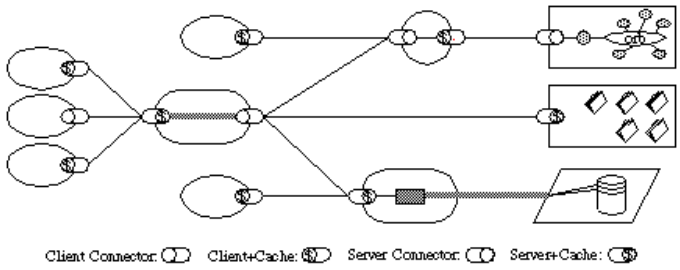


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

There are substantial security benefits of having a layered system since it allows you to stop attacks at the proxy layer, or within other layers, preventing them from getting to your actual server architecture. By utilizing a layered system with a proxy, or creating a single point of access, you are able to keep critical and more vulnerable aspects of your architecture behind a firewall, preventing direct interaction with them by the client.

Keep in mind that security is not based on single “stop all” solution, but rather on having multiple layers with the understanding that certain security checks may fail or be bypassed. As such, the more security you are able to implement into your system, the more likely you are to prevent damaging attacks.

Code on Demand

Perhaps the least known of the six constraints, and the only optional constraint, Code on Demand allows for code or applets to be transmitted via the API for use within the application. In essence, it creates a smart application that is no longer solely dependent on its own code structure.

However, perhaps because it’s ahead of its time, Code on Demand has struggled for adoption as Web APIs are consumed across multiple languages and the transmission of code raises security questions and concerns. (For example, the directory would have to be writeable, and the firewall would have to let what may normally be restricted content through.)

Just the same, I believe that while it is currently the least adopted constraint because of its optional status, we will soon see more implementations of Code on Demand as its benefit becomes more apparent.

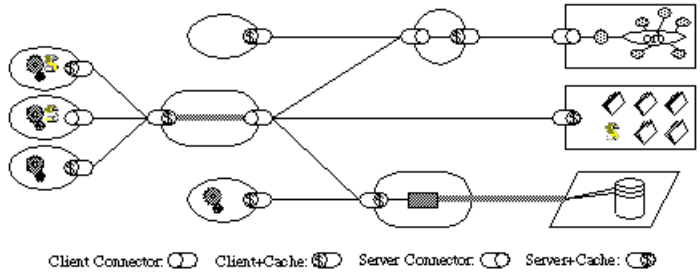


Figure 5-8. REST

Together, these constraints make up the theory of Representational State Transfer, or REST. As you look back through these you can see how each successive constraint builds on top of the previous, eventually creating a rather complex—but powerful and flexible—application program interface.

But most importantly, these constraints make up a design that operates similarly to how we access pages in our browsers on the World Wide Web. It creates an API that is not dictated by its architecture, but by the representations that it returns, and an API that—while architecturally stateless—relies on the representation to dictate the application's state.

Please keep in mind that this is nothing more than a quick overview of the constraints of REST as defined by Dr. Fielding. For more information on the different constraints, you can read his full dissertation online at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Architectural Diagrams provided courtesy of Roy Fielding, from *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

2

Planning Your API

While understanding which type of API you are building is a vital step in creating the perfect one for your company—and one that users will love—it is just as important to carefully plan out your API's capabilities.

Surprisingly, while being one of the most crucial steps in API development, this step is usually rushed through by companies excited to generate a product map and start working on code.

In many ways, building an API is like building a house. You can say, "I want my house to look like this picture," but without the blueprints, chances are it isn't going to turn out exactly the way you had hoped. Yet while we carefully plan and build houses, APIs have been plagued by the "agile methodology."

While I am a fan of agile and I find it to be one of the greatest advancements in project management, like all good things, it has its time and place. Unlike many Software as a Service applications built for today's web, an API's interface is a contract, and as such cannot be constantly changing. With increases in the use of hypertext links and hypermedia,

perhaps some day in the future that may no longer be true, but right now many developers are hardcoding resources and actions, and as such changing the resource name, moving it or changing the data properties can be detrimental to their application.

It's also important to understand that developers won't just be relying on your API to do cool things—they'll also be relying on it for their livelihood. When you put out an API, you are giving developers your company's word, and when you break the API, you are not just breaking your company's word, you're undermining their ability to provide for their families. So it's important that you create a solid foundation for your API, just as you would a house. It's important that you understand what your API should be able to do, and how it will work. Otherwise, you may lock yourself into a poor design or worse—create an API that doesn't meet your developers' needs. If that happens, you'll find yourself taking a lot more time and spending a lot more money to fix what you should have planned up front instead.

Who is Your API For

As developers, we tend to like to jump to what our API *will* do before we think about what our API *should* do, or even what we *want* it to do. So before we get started, we need to take a step back and ask ourselves, "Who will be using our API?"

Are you building your API for your application's customers? For their business partners? For third party developers so that your platform can be extended upon? Often times the answer tends to be a combination of the above, but until you understand *for whom* you are building your API, you aren't ready to start planning it.

To Which Actions Do They Need Access?

All too often I hear companies say, "We want to build an API to expose our data," or "We want to build an API to get people using our system." Those

are great goals, but just because we want to do something doesn't mean we can truly accomplish it without a solid plan.

After all, we all want people to use our APIs, but why should they? While it may seem a little harsh, this is the very next question we need to carefully ask ourselves: Why would our users (as we identified them above) want to use our API? What benefit does it offer them?

Another common mistake is answering the question of "why?" with, "our reputation." Many companies rely on their name rather than their capabilities. If you want to grow a strong and vibrant developer community, your API has to do something more than just bear your name.

What you *should* be doing is asking your potential users, "Which actions would you like to be able to accomplish through an API?" By speaking directly to your potential API users, you can skip all the guesswork and instead find out exactly what they are looking for and which actions they want to be able to take within your API, and also isolate your company's value to them. Often, the actions your users will want are different from the ones you anticipated, and by having this information you can build out your API while testing it for *real* use cases.

Remember, when building an application for a customer, we sit down either with them or the business owners to understand what it is they want us to build. Why aren't we doing the same thing with APIs? Why aren't we sitting down with our customers, the API users, and involving them in the process from day one? After all, doing so can save us a lot of time, money and headaches down the road.

Plus, by asking your users what they want, you may find you have better selling points to get the support from other areas within the business. Most importantly, you can answer the vital question of *why* you are building the API in the first place.

List out the Actions

Now that you know what your developers want to do, list out the actions. All too commonly, developers jump into the CRUD mindset, but for this exercise, you should simply create categories and add actions to them based on what part of the application or object it will affect.

For example, your developers will probably want access to their users, the ability to edit a user, reset a password, change permissions, and even add or delete users.

Assuming your application has a messaging system, they may also want to create a new draft, message a user, check messages, delete messages, etc.

As you isolate these different categories and actions, you'll want to chart them like so:

Users	Create a user, edit a user, retrieve username, retrieve password, reset password, view profile, **Message user**
Messages	**Send a message**, create a draft, send a draft, delete draft, get message, mark message as read, mark message as unread, move message to folder, delete message
Products	View product, review product, **add product to cart,** add to wishlist
Cart	View cart, **add product**, change quantity, delete product, checkout

Now, you may notice that in the above chart we have duplicate entries. For example, we have "message a user" under both "Users" and "Messages." This is actually a good thing, because it shows us how the different actions work together across different categories, and potentially, different resources.

We now know that there is a viable use case for messages not only within users, but also within messages, and we can decide under which it makes the most sense. In the case of “send a message” it would probably make most sense under the “messages” resource, however because of the relationship, we might want to include a hypertext link when returning a user object in the “users” resource.

By doing this exercise, not only can we quickly isolate which actions we need to plan for, but also how they’ll work together and even how we should begin designing the flow of our API.

This step may seem incredibly simple, but it is one of the most crucial steps in the planning process to make sure you are accounting for your developers’ needs (I would even recommend showing your potential API users your chart to see if they think of anything else after the fact), while also understanding how the different resources within your API will need to work together, preventing you from having to rewrite code or try to move things around as you are coding your API.

Explain How Your API Will Interact with Existing Services

Unless you are starting from ground zero and taking an API-first approach, there’s a good chance you have other applications and services that your API may need to interact with.

You should take time to focus on how your API and application will interact. Remember, your application can change over time, but you’ll want your API’s interface to remain consistent for as long as possible.

You’ll also want to take time to isolate which services the API will need to interact with. Even before building the API you can ensure that these services are flexible enough to talk to your API while keeping it decoupled from your technology stack.

Along with understanding any technical risks involved with these services, if they are organizationally owned, developers can start focusing on transitioning them to an API-focused state, ensuring that by the time you are ready to connect them to your API, they are architecturally friendly. It is never too early to plan ahead.

How Are You Going to Maintain Your API?

Remember that building an API is a long-term commitment, because you won't just be creating it, you will be maintaining it as well.

It's very possible to build a good API that doesn't need much work after its release, but more typically than not, especially if you're an API-driven company, you'll find that not only are there bugs to fix, but developers' demands will increase more and more as they use your API for their applications.

One of the advantages to the Spec-Driven Development approach to APIs is that you start off by building the foundation of your API, and then slowly and carefully adding to it after that. This is the recommended approach, but regardless you shouldn't plan on just launching it and leaving it, but rather having dedicated resources that can continue to maintain, patch bugs, and hopefully continue to build upon your API as new features are released within your application.

How Are You Going to Version Your API

Along with maintaining your API, you should also plan on how you are going to version your API. Will you include versioning in the URL such as http://api.mysite.com/v1/resource, or will you return it in the content-type (application/json+v1), or are you planning on creating your own custom versioning header or taking a different approach altogether?

Keep in mind that your API should be built for the long-term, and as such you should plan on avoiding versioning as much as possible, however, more likely than not there will come a time when you need to break backwards incompatibility, and versioning will be the necessary evil that lets you do so.

We'll talk about versioning more, but in essence you should try to avoid it, while still planning for it—just as you would plan an emergency first aid kit. You really don't want to have to use it, but if you do – you'll be glad you were prepared.

How Are You Going to Document Your API

Along with maintenance, developers will need access to documentation, regardless if you are building a hypermedia driven API or not.

And while documentation may seem like a quick and easy task, most companies will tell you it is one of their biggest challenges and burdens when it comes to maintaining their API.

As you update your API you will want to update your documentation to reflect this, and your documentation should have a clear description of the resource, the different methods, code samples, and a way for developers to try it out.

We'll look at documentation more in-depth towards the end of this book, but fortunately the steps we take in the next chapter will help us efficiently create (and maintain) our documentation. Whether or not you elect not to follow these steps, you should have a plan for how you are going to maintain your API's documentation. This is one area you should not underestimate since it has proven to be the crux of usability for most public APIs.

How will Developers Interact with Your API

Another important aspect to plan for is how developers will interact with your API. Will your API be open like the Facebook Graph API, or will you utilize an API Key? If you're using an API key, do you plan on provisioning your API to only allow certain endpoints, or set limits for different types of users? Will developers need an access token (such as OAuth) in order to access user's data?

It's also important to think about security considerations and throttling. How are you going to protect your developer's data, and your service architecture? Are you going to try and do this all yourself, or does it make more sense to take advantage of a third-party API Manager such as MuleSoft?

The answers to these questions will most likely depend on the requirements that you have for your API, the layers of security that you want, and the technical resources and expertise you have at your company. Generally, while API Management solutions can be pricey, they tend to be cheaper than doing it yourself.

We'll talk more about these considerations and using a proxy or management solution in another chapter.

How Are You Going to Manage Support

Another consideration, along with documentation is how are you going to manage support when things go wrong? Are you going to task your engineers with API support questions? If so, be warned that while this may work in the short-term, it is typically not scalable.

Are you going to have a dedicated API support staff? If so, which system(s) will you use to manage tickets so that support requests do not get lost, can be followed up on, escalated, and your support staff do not lose their minds.

Are you going to have support as an add-on, a paid service, a partner perk, or will it be freely available to everyone? If support is only for certain levels or a paid subscription, will you have an open community (such as a forum) for developers to ask questions about your API, or will you use a third-party like StackOverflow? And how will you make sure their questions get answered, and bugs/ documentation issues get escalated to the appropriate source?

Don't Be Intimidated

In this chapter we've asked a lot of questions, hopefully some thought-provoking ones. The truth is that building and running a successful API is a lot of work, but do not let these questions scare you off.

The reason we ask these questions now is so that you are thinking about them, and so that you have the answers and can avoid surprises when the time comes. These questions are designed to prepare you, not overwhelm you, so that you can build a truly successful program and avoid costly mistakes. Because a strong API program doesn't just drive adoption, it drives revenue, and it drives the company forward.

> ***"I can definitely say that one of the best investments we ever made as a company was in our API. It's probably the best marketing we've ever done."***
>
> – Ben Chestnut, Owner of MailChimp

3

Designing the Spec

Once you understand why you are building your API, and what it needs to be able to accomplish you can start creating the blueprint or spec for your API. Again, going back to the building a house scenario, by having a plan for how your API should look structurally before even writing a line of code you can isolate design flaws and problems without having to course correct in the code.

Using a process called Spec-Driven Development, you will be able to build your API for the long-term, while also catching glitches, inconsistencies and generally bad design early on. While this process usually adds 2–4 weeks onto the development cycle, it can save you months and even years of hassle as you struggle with poor design, inconsistencies, or worse—find yourself having to build a brand-new API from scratch.

The idea behind a REST API is simple: it should be flexible enough to endure. That means as you build your API, you want to plan ahead—not just for this development cycle, not just for the project roadmap, but for what may exist a year or two down the road.

This is really where REST excels, because with REST you can take and return multiple content types (meaning that if something comes along and replaces JSON, you can adapt) and even be fluid in how it directs the client with hypermedia. Right off the bat you are being setup for success by choosing the flexibility of REST. However it's still important that you go in with the right mindset—that the API you build will be long-term focused.

Versioning – A Necessary Evil

As previously mentioned, versioning is important to plan for, but all too often companies look at an API the same way they do desktop software. They create a plan to build an API—calling it Version 1—and then work to get something that's just good enough out the door. But there's a huge difference between creating a solid foundation that you can add onto and a half-baked rush job just to have something out there with your name on it. After all people will remember your name for better or worse.

The second problem is they look at versions as an accomplishment. I remember one company that jumped from Version 2 to Version 10 just because they thought it sounded better and made the product look more advanced. But with APIs, it's just the opposite. A good API isn't on Version 10, it's still rocking along at Version 1, because it was that well thought out and designed in the first place.

If you go into your API design with the idea that you are "only" creating Version 1, you will find that you have done just that—created a version that will be short lived and really nothing more than a costly experiment from which you hopefully learned enough to build your "real" API. However, if you follow the above steps and carefully craft your API with the idea that you are building a foundation that can be added onto later, and one that will be flexible enough to last, you have a very good chance of creating an API that lives 2–3 years—or longer!

Think about the time and cost it takes to build an API. Now think about the time it takes to get developers to adopt an API. By creating a solid API now, you avoid all of those costs upfront. And in case you're thinking, "It's no big deal, we can version and just get developers to upgrade," you might want to think again. Any developer evangelist will tell you one of the hardest things to do is to get developers to update their code or switch APIs. After all, if it works, why should they change it? And remember, this is their livelihood we're talking about—they can spend time making money and adding new features to their application, or they can spend time losing money trying to fix the things you broke—which would you prefer to base your reputation upon?

Versioning an API is not only costly to you and the developer, it also requires more time on both ends, as you will find yourself managing two different APIs, supporting two different APIs, and confusing developers in the process. In essence, when you do version, you are creating the perfect storm.

You should NOT version your API just because you've:

- Added new resources
- Added data in the response
- Changed technologies (Java to Ruby)
- Changed your application's services

Remember, your API should be decoupled from both your technology stack and your service layer so that as you make changes to your application's technology, the way the API interacts with your users is not impacted. Remember the uniform interface—you are creating separation between your API and your application so that you are free to develop your application as needed, the client is able to develop their application as needed, and both are able to function independently and communicate through the API.

However, you SHOULD consider versioning your API when:

- You have a backwards-incompatible platform change, such as completely rewriting your application and completely changing the user interface
- Your API is no longer extendable—which is exactly what we are trying to avoid here
- Your spec is out of date (e.g. SOAP)

Again, I cannot stress enough the importance of going into your API design with a long-term focus, and that means versioning becomes a last-resort or doomsday option for when your API can no longer meet your users'—or the company's—needs.

Understand You're Poor at Design

The next thing that's important to understand is that we, as developers, are poor at long-term design.

> ***"People are fairly good at short-term design and usually awful at long-term design."***
>
> – Dr. Roy Fielding, Creator of REST

Think about a project you built three years ago, even two years ago, even last year. How often do you start working on a project only to find yourself getting stuck at certain points, boxed in by the very code you wrote? How often do you look back at your old code and ask yourself, "What was I thinking?" I don't know about you, but I do that on an almost daily basis (and usually with yesterday's—or this morning's—code).

The simple fact is that we can only see what we can see. While we may think we are thinking through all the possibilities, there's a good chance we're missing something. I can't tell you how many times I've had the chance to do peer-programming where I would start writing a function or method, and the other developer would ask why I didn't just do it in two lines of code instead!? Of course their way was the right way, and super simple, but my mind was set, and I had developer tunnel vision—something we all get that is dangerous when it comes to long-term design.

By accepting that we, by ourselves, are not good at long-term design, we actually enable ourselves to build better designs. By understanding that we are fallible and having other developers look for our mistakes (in a productive way), we can create a better project and a longer-lasting API. After all, two heads are better than one!

In the past this has been difficult, especially with APIs. Companies struggle to afford (or even recognize the benefit of) peer programming, and building out functional mock-ups of an API has proven extremely costly. Thankfully, advances in technology have made it possible to get feedback—not just from our coworkers, but also from our potential API users—without having to write a single line of code! This means that where before we would have to ship to find inconsistencies and design flaws, now we can get feedback and fix them before we even start coding our APIs, saving time and money not only in development, but also support.

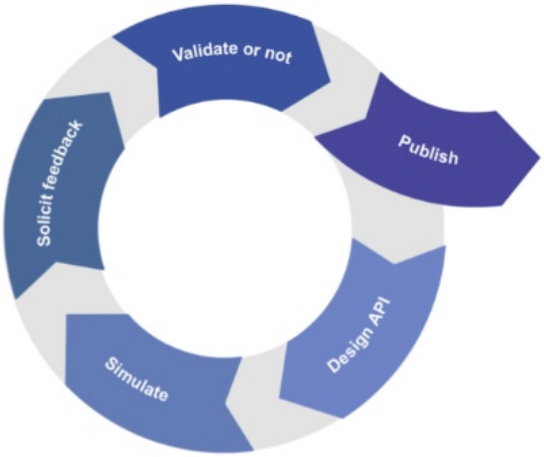
To take advantage of this new technology to the fullest, we can use a methodology that I am calling Spec-Driven Development, or the development of our API based on a predefined specification that has been carefully tested and evaluated by our potential API users.

Spec-Driven Development

Spec-Driven Development is designed to take advantage of newer technologies in order to make the development, management and documentation of our API even more efficient. It does this by first dividing design and development into two separate processes.

The idea behind Spec-Driven Development is that agility is a good thing, and so is agile user testing/ experience. However, what we do not want to see is agile development of the API design when it comes time to write the code. Instead, Spec-Driven Development encourages separating the design stage from the development stage, and approaching it iteratively. This means that as you build out your design using a standardized spec such as RESTful API Modeling Language (RAML), you can test that spec by mocking it up and getting user feedback.

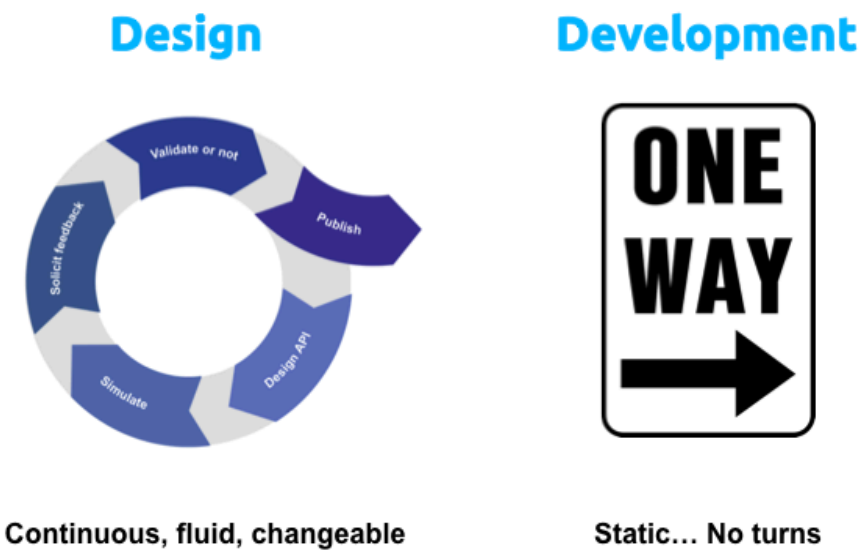
MuleSoft's API Contract Design Cycle demonstrates a well-thought-out flow for perfecting your spec. It begins with designing your API, then moves to mocking/simulating the API, soliciting feedback, and finally—depending on that feedback—either determining that the spec is ready for development or returning to the design stage where the cycle continues.



Once you have finished getting user feedback and perfecting the design in the spec, then you can use that specification as your blueprint for design.

In essence, you are keeping agile user testing, and agile development, but splitting them so you're not doing agile user testing as you do the actual development (as your code should be written to match the spec's design, and thus the previously tested and affirmed user experience).

It's important to note that with Spec-Driven Development, there is no back and forth. Once you move into the development phase, you are moving forward with the assumption that the spec has been perfected, and that you have eliminated 99 percent of the design flaws/ inconsistencies. Should you find an issue with the design, rather than correcting it in the development cycle, you need to stop and go back to the design cycle, where you fix the spec and then retest it.



The reasoning behind this is pretty simple—we're usually awful at long-term design. As such, rather than make changes on the fly or try to fix things with a short-sighted view, Spec-Driven Development encourages "all hands

on deck" to ensure that the changes you make (no matter how simple or insignificant they might seem) do not cause any design inconsistencies or compounding problems, either in the short-term or down the road.

In order to be successful with Spec-Driven Development, you should follow these six constraints:

1. Standardized
Spec-Driven Development encourages the use of a standardized format applicable to the type of application you are building. In the case of building an API, for example, the following specs would be considered standard or common among the industry: RAML, Swagger, API Blueprint, IO Docs. Utilizing a standard spec ensures easy portability among developers while also ensuring that the spec your application relies on has been thoroughly tested by the community to ensure that it will meet both your short-term and long-term needs while maintaining consistency in its own format.

2. Consistent
In developing your spec, you should utilize pattern-driven design as well as code reuse when possible to ensure that each aspect of your spec is consistent. In the event of building an API, this would mean ensuring your resources are all formatted similarly and your methods all operate in a similar format—both in regards to the request and available responses. The purpose of consistency is to avoid confusion in both the development and use of your application so all aspects of the application work similarly, providing the end user with the freedom to move seamlessly from one focus to another.

3. Tested
Spec-Driven Development requires a strong, tested spec in order to build a reliable application. This means that the spec has to be carefully crafted and then tested with both internal and external uses to ensure that it accomplishes its goals and meets the needs of all parties.

The spec should be crafted, mocked/prototyped and tested to retrieve user feedback. Once user feedback is received, the spec should be modified appropriately, mocked and tested again, creating a continuous cycle until you have perfected the spec—or at the least eliminated a large majority of the design issues to ensure spec and application longevity.

4. Concrete

The specification should be the very foundation of your application or, in essence, the concrete foundation of the house you are building. The spec should encompass all aspects of your application, providing a solid blueprint to which your developers can code. The spec does not have to encompass future additions, but it should have taken as many of them into consideration as possible. However, there is nothing that relates to the spec that is coded outside of existing inside of the spec.

5. Immutable

The spec is the blueprint for development and is unchangeable by code. This means that at no time is the code to deviate from or override the spec. The spec is the ultimate authority of the application design, since it is the aspect that has been most thought out and carefully designed, and has also been tested by real-world users. It is important to realize that short-term coding implementations can be detrimental to an application's longevity, and as such have no place in Spec-Driven Development.

6. Persistent

All things evolve, and the application and spec are no different. However, each evolution must be just as carefully thought out as the original foundation. The spec can change, but each change must be justified, carefully evaluated, tested and perfected. In the event of redevelopment, if the spec is found not to be renderable, it is important

to go back and correct the spec by re-engaging in user testing and validation, and then updating the code to match to ensure that it is consistent with your spec, while also ensuring that the necessary changes do not reduce the longevity of your application.

The last constraint of Spec-Driven Development, the Persistent constraint, explains how you can use Spec-Driven Development to continue building out and adding additions to your API. For every change you make to the API, you should start with the design stage, testing your new additions for developers to try out, and then once validated, start adding the code and pushing the changes to production. As described above in the Immutable constraint, your API should never differ from the spec or have resources/methods that are not defined in the spec.

Along with helping ensure that your API is carefully thought out, usable and extendable for the long-term, use of these common specs offers several additional benefits, including the ability auto generate documentation and create interactive labs for developers to explore your API. There are even Software Development Kit (SDK) generation services such as APIMatic.io and REST United that let you build multi-language SDKs or code libraries on the fly from your spec!

This means that not only have you dramatically reduced the number of hours it will require you to fix bugs and design flaws, handle support or even rebuild your API, but you are also able to cut down on the number of hours you would be required to write documentation/create tools for your API, while also making your API even easier for developers to use and explore.

Of course, this leads us to choosing the right spec for the job.

Choosing a Spec

Choosing the best specification for your company's needs will make building, maintaining, documenting and sharing your API easier. Because Spec-Driven Development encourages the use of a well tested, standardized spec, it is highly recommended that you choose from RAML, Swagger or API Blueprint. However, each of these specs brings along with it unique strengths and weaknesses, so it is important to understand what your needs are, as well as which specification best meets those needs.

RAML

The baby of the three most popular specs, RAML 0.8, was released in October 2013. This spec was quickly backed by a strong working group consisting of members from MuleSoft, PayPal, Intuit, Airware, Akana (formally SOA Software), Cisco and more. What makes RAML truly unique is that it was developed to model an API, not just document it. It also comes with powerful tools including a RAML/ API Designer, an API Console and the API Notebook—a unique tool that lets developers interact with your API. RAML is also written in the YAML format, making it easy to read, and easy to edit—regardless of one's technical background.

Swagger

The oldest and most mature of the specs, Swagger, just recently released Version 2.0, a version that changes from JSON to the YAML format for editing and provides more intuitive tooling such as an API Designer and an API Console. Swagger brings with it the largest community and has recently been acquired by SmartBear, while also being backed by Apigee and 3Scale. However, with the transition from JSON to YAML, you may find yourself having to maintain two different specs to keep documentation and scripts up to date. Swagger also lacks strong support for design patterns and code reusability—important aspects of Spec-Driven Development.

API Blueprint

Created by Apiary in March of 2013, API Blueprint is designed to help you build and document your API. However, API Blueprint lacks the tooling and language support of RAML and Swagger, and utilizes a specialized markdown format for documenting, making it more difficult to use in the long run. Just the same, API Blueprint has a strong community and does excel with its documentation generator.

Overall, RAML offers the most support for Spec-Driven Development and provides interactive tools that are truly unique. You can also use's free mocking service to prototype your API instead of having to install applications and generate the prototype yourself - as you currently have to do with Swagger. On the other hand, Swagger offers the most tooling and largest community.

Attribute	API Blueprint	RAML	Swagger
Usability			
Spec	★★★	★★★★★	★★★★
Website	★★★★★	★★★★	★★★★★
Language Support	★	★★★	★★★★★
Ease of Getting Started	★★	★★★★★	★★★★
Ease of Spec Maintenance	★★★★★	★★★★★	★★★
Spec Driven			
API Design First	Not Yet Rated	★★★★★	★★★
Visual API Designer	N/A	★★★★★	★★★★★
Allows Code Reuse	Not Yet Rated	★★★★	★
Mocking Service	★★	★★★★	★★
Build Support			
Frameworks	★	★	★★★★
Applications/ Tools	★	★★	★★
Support Tooling			
API Console	N/A	★★★★	★★★
API Documentation	★★★★★	★★★	★★★
API Interaction	N/A	★★★★★	N/A
API SDK Generation	★★	★★	★★★
Community			
Size	★★★	★★★	★★★★★
Engagement	★★★★★	★★★★★	★★★★★
Available Resources	★	★★	★★★★★
Industry Backing	★	★★★★★	★★

Be sure to take a look at the chart on the previous page to see which specification best matches your business needs. However, I would personally recommend that unless you have needs that cannot be met by RAML, that you strongly consider using this spec to define your API. Not only will you have the ability to reuse code within your spec, but RAML offers the strongest support for Spec-Driven Development, has strong support for documentation and interaction, has unique, truly innovative tools and offers strong support for SDK generation and testing.

In the next chapter we'll take a look at using RAML, but again, this is only a recommendation. What is most important is that you choose the spec that meets your individual needs, as these three specs all have their own strengths and weaknesses.

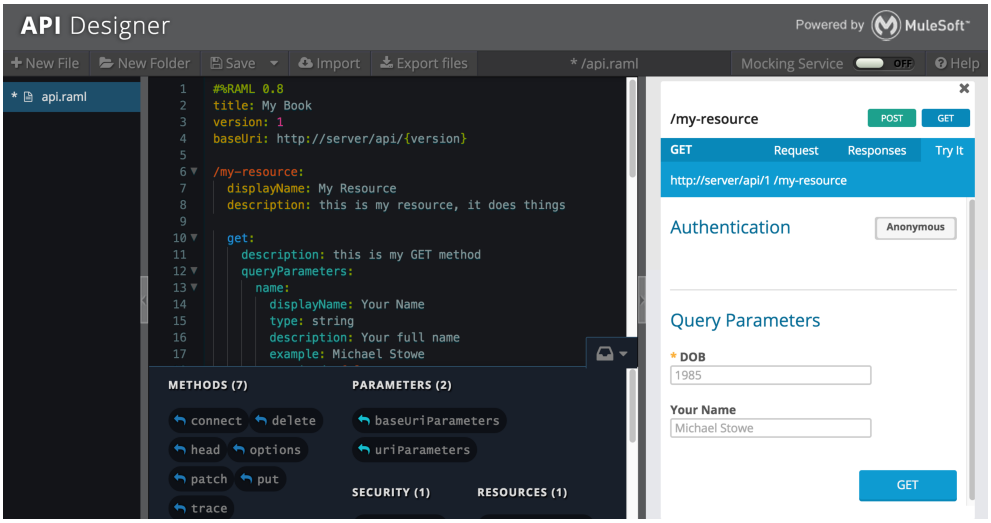
4

Using RAML

One of the easiest ways to start working with RAML is with the API Designer, a free open source tool available on the RAML website at http://raml.org/projects.

To get started even faster, MuleSoft also offers a free, hosted version of its API Designer. . You can take advantage of this free service by visiting https://anypoint.mulesoft.com/apiplatform/.

Because RAML is defined in the YAML (Yet Another Markup Language) format, it is both human- and machine-readable, and with the API Designer you will receive auto suggestions, tooltips, and available options, while also having a visual preview to the right of your screen.



RAML requires that every API have a title, a version and a baseUri. These three aspects help ensure that your API can be read, versioned and accessed by any tools you choose to implement.

Describing these in RAML is as easy as:

```
#%RAML 0.8
title: My Book
version: 1
baseUri: http://server/api/{version}
```

The nice thing is that the API Designer starts off by providing you the first three lines, so all you need to add is your baseUri. You'll also notice that RAML has a version placeholder, letting you add the version to the URI if desired.

To add a resource to your RAML file, simply declare the resource by using a slash "/" and the resource name followed by a colon ":" like so:

```
#%RAML 0.8
title: My Book
version: 1
baseUri: http://server/api/{version}

/my-resource:
```

YAML is tab-delimited, so once we have declared /my-resource, we can set the properties of the resource by tabbing over once.

```
#%RAML 0.8
title: My Book
version: 1
baseUri: http://server/api/{version}

/my-resource:
   displayName: My Resource
   description: this is my resource, it does things
```

To add a method, such as GET, POST, PUT, PATCH or DELETE, simply add the name of the method with a colon:

```
#%RAML 0.8
title: My Book
version: 1
baseUri: http://server/api/{version}

/my-resource:
   displayName: My Resource
   description: this is my resource, it does things

   GET:

   POST:
```

You can then add descriptions, query parameters, responses with examples and schemas, or even additional nested endpoints, letting you keep all of your resources grouped together in an easy-to-understand format:

```
#%RAML 0.8
title: My Book
version: 1
baseUri: http://server/api/{version}

/my-resource:
   displayName: My Resource
   description: this is my resource, it does things

   get:
      description: this is my GET method
          queryparameters:
          name:

      responses:
         200: …

      post:
      description: this is my post method

   /sub-resource:
      displayName: Child Resource
      description: this is my sub resource
```

URI Parameters

Because resources often take dynamic data, such as an ID or even a search filter, RAML supports URI Parameters/placeholders. To indicate

dynamic data (which can either be defined by RAML or the URI), just use the braces as you did with {version} in the baseUri:

```
#%RAML 0.8
title: My Book
version: 1
baseUri: http://server/api/{version}

/my-resource:
   /sub-resource/{id}:

   /{searchFilter}:
```

Query Parameters

As shown above, you can easily add query parameters or data that is expected to be sent to the server from the client when making an API call on that resource's method. RAML also lets you describe these parameters and indicate whether or not they should be required:

```
/my-resource:
   get:
      queryParameters:
         name:
            displayName: Your Name
            type: string
            description: Your full name
            example: Michael Stowe
            required: false

         dob:
            displayName: DOB
            type: number
            description: Your date of birth
            example: 1985
            required: true
```

As you've probably noticed, RAML is very straight-forward and uses descriptive keys. (displayName) for how you want it to be displayed in the documentation, a description to describe what it does, the type (string, number, etc), an example for the user to see, and whether or not the parameter is required (boolean).

Responses

Likewise, RAML tries to make documenting method responses fairly straight-forward by first utilizing the responses key and then displaying the type of responses a person might receive as described by their status code. (We'll look at the different codes later.) For example, an OK response has status code 200, so that might look like this:

```
/my-resource:
   get:
      responses:
         200:
```

Within the 200 response we can add the body key to indicate the body content they would receive back within the 200 response, followed by the content-type (remember APIs can return back multiple formats), and then we can include a schema, an example, or both:

```
/my-resource:
   get:
      responses:
         200:
            body:
               application/json:
                  example: |
                  {
                   "name" : "Michael Stowe",
```

```
 "dob" : "1985",
 "author" : true
}
```

To add additional content-types we would simply add a new line with the same indentation as the “application/json” and declare the new response type in a similar fashion (e.g.: application/xml or text/xml).

To add additional responses, we can add the response code with the same indentation as the 200, using the appropriate status code to indicate what happened and what they will receive.

As you are doing all of this, be sure to look to the right of your editor to see your API being formed before your very eyes, letting you try out the different response types and results.

ResourceTypes

As you can imagine, there may be a lot of repetitive code, as you may have several methods that share similar descriptions, methods, response types (such as error codes) and other information. One of the nicest features in RAML is resourceTypes, or a templating engine that lets you define a template (or multiple templates) for your resource to use across the entire RAML spec, helping you eliminate repetitive code and ensuring that all of your resources (as long as you use a standardized template/ resourceType) are uniform.

```
resourceTypes:
 - collection:
      description: Collection of available
   <<resourcePathName>>
      get:
         description: Get a list of <<resourcePathName>>.
         responses:
```

```
        200:
            body:
                application/json:
                    example: |
                        <<exampleGetResponse>>

        301:
            headers:
                location:
                    type: string
                    example: |
                        <<exampleGetRedirect>>
        400:

/my-resource:
   type:
      collection:
         exampleGetResponse: |
            {
             "name" : "Michael Stowe",
             "dob" : "1985",
             "author" : true
            }
         exampleGetRedirect: |
            http://api.mydomain.com/users/846

/resource-two:
   type:
      collection:
         exampleGetResponse: |
            {
             "city" : "San Francisco",
             "state" : "1985",
             "postal" : "94687"
```

```
        }
    exampleGetRedirect: |
        http://api.mydomain.com/locations/78
```

In the above example we first define the resourceType “collection,” and then call it into our resource using the type property. We are also taking advantage of three placeholders <<resourcePathName>> that are automatically filled with the resource name (“my-resource,” “resource-two”), and <<exampleGetResponse>> and <<exampleGetRedirect>>, which we defined in our resources. Now, instead of having to write the entire resource each and every time, we can utilize this template, saving substantial amounts of code and time.

Both “my-resource” and “resource-two” will now have a description and a GET method with 200, 301 and 400 responses. The 200 response returns back an application/json response with the example response we provided using the <<exampleGetResponse>> placeholder, and a redirect in the case of a 301 with <<exampleGetRedirect>>.

Again, we will get all of this without having to write repetitive code by taking advantage of resourceTypes.

Traits

Like resourceTypes, traits allow you to create templates, but specifically for method behaviors such as isPageable, isFilterable and isSearchable.

```
traits:
   -searchable:
        queryParameters:
            query:
                description: |
                    JSON array
```

```
                    [{"field1","value1","operator1"},…]
                    <<description>>
                example: |
                    <<example>>
/my-resource:
   get:
   is: [searchable: {description: "search by location
   name", example: "[\"city\"\,\"San Fran\",\"like\"]"}]
```

To utilize traits, we first define the trait that we want to use, in this case “searchable” with the query parameters that we want to use, including the description (using the <<description>> placeholder) and an example (using the <<example>> placeholder).

However, unlike with resourceTypes, we pass the values for these placeholders in the searchable array within the “is” array (which can hold multiple traits).

Again, like resourceTypes, traits are designed to help you ensure that your API is uniform and standard in its behaviors, while also reducing the amount of code you have to write by encouraging and allowing code reuse.

Going Further

Hopefully this has given you a good start with RAML, but there are many more tools and features to make defining your API even easier (including easy implementation of common authorization methods such as basic auth and OAuth1 & 2). Be sure to go through the RAML 100 and 200 Tutorials on the RAML website (http://raml.org), and take a look at the RAML spec (http://raml.org/spec.html) to learn even more pro tips and tricks.

Now, once you have your API defined, we can quickly prototype it using the same MuleSoft tools, getting feedback from our potential users and identifying potential design flaws and bugs!

5

Prototyping & Agile Design

As you design your spec, one of the most important things you can do is involve your users, getting crucial feedback to ensure it meets their needs, is consistent and is easily consumable.

The best way to do this is to prototype your API and have your potential users interact with it as if it was the actual API you are building. Unfortunately, until recently this hasn't been possible due to constraints in time and budget resources. This has caused companies to utilize a "test it as you release it" method, where they build the API to what they think their users want, and after doing internal QA, release it in the hope that they didn't miss anything. This Wild West style of building APIs has led to numerous bugs and inconsistencies, and greatly shortened API lifetimes.

Thankfully, RAML was designed to make this process extremely simple, allowing us to prototype our API with the click of a button, creating a mock

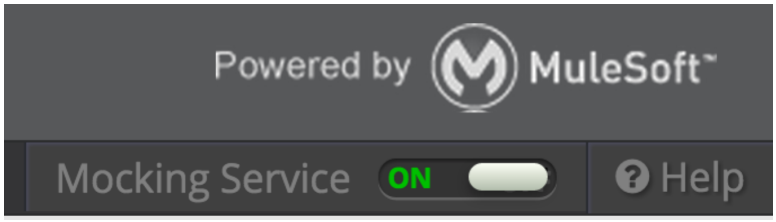
API that relies on example responses that can be accessed from anywhere in the world by our users.

Likewise, Swagger and API Blueprint offer some building and mocking tools, however, right now there isn't anything quite as simple or easy to use as MuleSoft's free mocking service.

Mocking Your API

MuleSoft's API designer not only provides an intuitive way to visually design your API, as well as interact and review resources and methods for completeness/documentation purposes, but it also provides an easy toggle to quickly build a hosted, mocked version of your API that relies on the "example" responses.

To turn on the Mocking service, one only needs to click the "Mocking Service" toggle switch into the "On" setting:



When set to "On" MuleSoft's free API Designer will comment out your current baseUri and replace it with a new, generated one that may be used to make calls to your mock API.

```
1  #%RAML 0.8
2  #baseUri: http://server/api/{version}
3  baseUri: http://mocksvc.mulesoft.com/mocks/c77e028b-cfd5-
   407d-aac4-69f8a4102687
4  title: My Amazing API
5  version: 1
```

This new baseUri is public, meaning that your potential users can access it anywhere in the world, just as if your API was truly live. They will be able to make GET, POST, PUT, PATCH, DELETE and OPTIONS calls just as they would on your live API, but nothing will be updated, and they will receive back example data instead.

MuleSoft's mocking service currently supports RAML and Swagger—although when importing Swagger it is converted to RAML.

Again, what makes prototyping so important is that your users can actually try out your API before you even write a line of code, helping you catch any inconsistencies within the API, such as inconsistencies in resource naming, method interaction, filter interactions or even in responses:

```
Endpoint /user:

{
     "id" : "1",
     "name" : "John"
     "birthday" : "01/01/1970",
     "emailId" : "2"
}
```

```
Endpoint /special:

{
     "data" : {
          "userId" : "1",
          "name" : "John"
          "dob" : "01/01/1970",
          "email" : "john@mulesoft.com"
     }
}
```

After all, as you build out your API, you want all of your data to be consistent, and you want to ensure that all of your user interactions are uniform, letting developers quickly utilize your API without having to isolate and debug special use cases where they need to do something unique or different just for that one particular case.

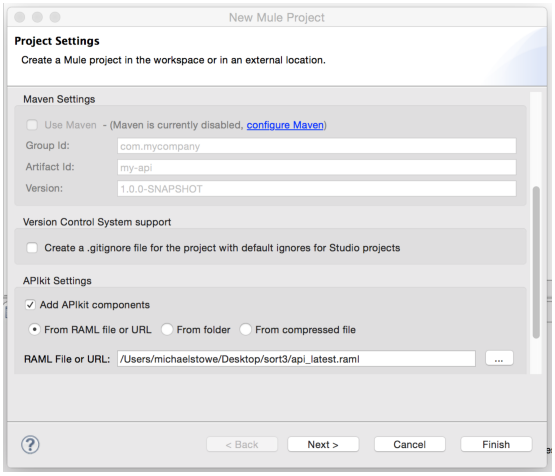
Alternative Mocking Methods

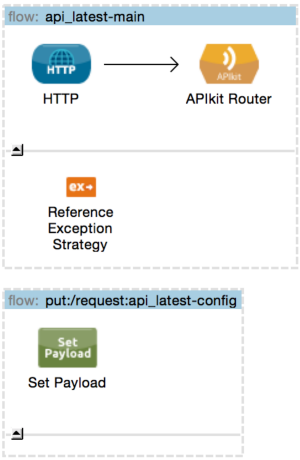
While the MuleSoft API Designer is a completely free, SaaS-hosted tool with no lock-in (there's no subscription or anything you have to buy), there are other ways to mock your RAML API.

Mockable.io and Apiary both provide tools that let you build out your API with a step-by-step (resource-by-resource) creation tool that creates a mock that your users can interact with. Mockable.io is also working on supporting spec importing (hoping to support RAML, Swagger, API Blueprint, Mashery IO Docs and WADL) but at the time of this writing, none of these imports were functional. Apiary also lets you import your API Blueprint spec.

Swagger also provides multiple clients for building out mocks of your API, although these clients must be downloaded and then hosted by you.

RAML specs can also be mocked out and hosted manually by using MuleSoft's Anypoint Studio with APIKit. This may be especially useful if you are also planning on building out your API with such a tool and need to integrate with multiple services, databases and third-party APIs.





But again, this requires not only downloading software and then either installing a runtime on your own server or hosting it on CloudHub, it also requires a manual process to generate the mock in the first place.

Last, but certainly not least, there are multiple language-based frameworks that allow the importing of specs, which we won't cover here.

Keep in mind that when choosing a service that is manually driven (not a SaaS such as MuleSoft, Apiary or Mockable.io) you are responsible for keeping the mock up to date with the spec—an exercise that proves to be both tedious and time-consuming.

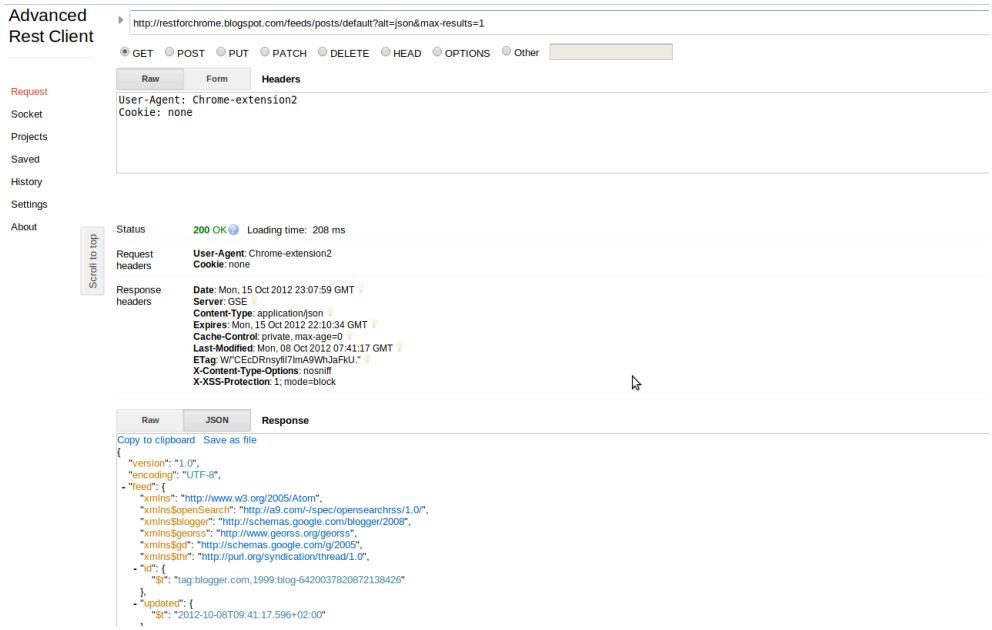
Getting Developers Involved

One of the hardest parts of the mocking/prototyping process is getting developers involved. Or as Gareth Jones, the Principle API Architect at Microsoft OneNote, points out:

> ***"Developers do not want to code against an API that is just going to be thrown away or might change the next day."***
>
> – Gareth Jones, Principle API Architect, Microsoft OneNote

Instead, to get developers involved with the testing and evaluation of your prototype, you have to make the process as simple as possible. This means providing your potential users with tools that let them interact with your API *without* having to write code.

For some, this may be as easy as pointing them to tools like Google Chrome's Advanced REST API Client (http://bit.ly/chromeRestAPIClient) that lets developers make REST calls right from their browser.

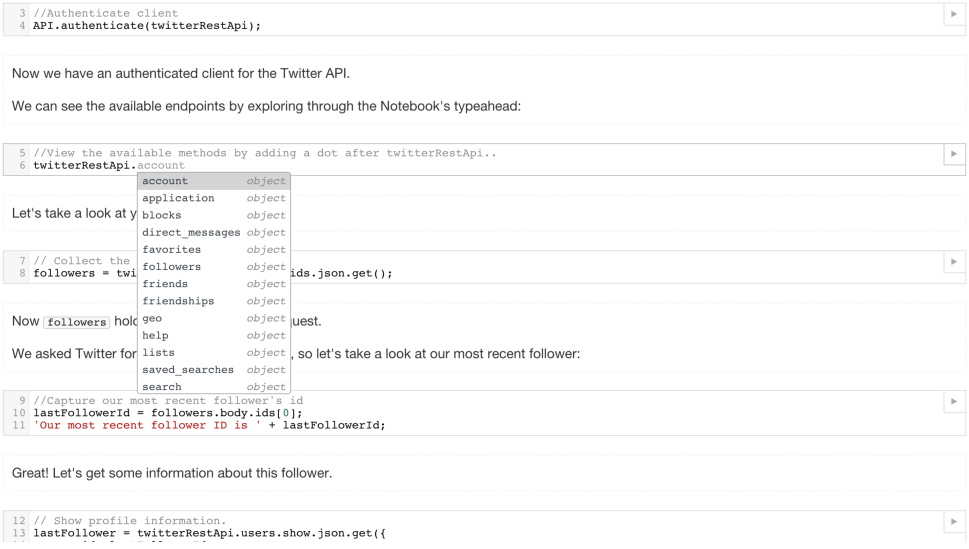


However, this tool still requires an in-depth introduction of your API to developers, something that can be easily provided with MuleSoft's API Portal and API Console that generates interactive documentation from the RAML spec but still requires developers to spend the time reading through these docs and manually transition from one call to another.

Ideally, you will want to have comprehensive documentation on your mock API for developers to access (again, something that can be easily generated from the RAML spec), but you do not want this to be the primary way your developers interact with and explore your API. Instead, the documentation should act as a supplementary tool, so that as developers want to do more complex calls or better understand your API, they can quickly access the information. But to get to this point, you need a quick on-ramp tool that will help teach developers what your API does and how they can use it—in five minutes or less.

One of the most useful tools for prototyping, as well as “teaching” purposes, is MuleSoft’s RAML-based API Notebook (http://www.apinotebook.com). The API Notebook lets you create API experiences through markup that developers can then edit and try out in real time without having to write any code. But to do more advanced calls, manipulate data and even have it interact with other data/APIs, the user just needs to know some basic JavaScript.

The API Notebook also lets users explore multiple resources by guiding them through all of the available options. It also lets them clone or create their own Notebooks that they can then share with you, providing you with valuable examples of real-use cases, bugs and inconsistencies that you can try out for yourself.



This also keeps developers from having to write advanced code to try out your API for their custom-use cases, and from having to share proprietary code either during the prototype or production phases!

Currently, the API Notebook is only available for RAML, and so far I have been unable to find an equivalent tool for Swagger or API Blueprint.

Getting Feedback

Once you provide your potential API users with a prototype and the tools to try it out, the next step is to provide a simple way for them to give you feedback. Ideally, during this stage you'll have a dedicated resource such as an API engineer or a Project Manager that can interact with your testers to not only get their feedback, but also have conversations to fully understand what it is that they are trying to do, or what it is that they feel isn't as usable as it should be. Keep in mind you'll also want to encourage your testers to be as open, honest and blunt as possible, as they may try to be supportive by ignoring issues or sugarcoating the design flaws that bother them at first—a kind but costly mistake that will ultimately harm both you and your potential users.

This step provides two valuable resources to the company. First, it provides a clearer understanding of *what* it is you need to fix (sometimes the problem isn't what a person says, but rather what the person is trying to do), while also telling your users that you listen to them, creating ownership of your API.

Many companies talk about creating a strong developer community, but the simplest way is to involve developers from day one. By listening to their feedback (even if you disagree), you will earn their respect and loyalty—and they will be more than happy to share your API with others, since they will be just as proud of it as you are.

It's also important to understand that people think and respond differently. For this reason you'll want to create test cases that help your testers understand what it is you are asking of them. However, you should not make them so restrictive or "by the book" that testers cannot veer off course and try out "weird" things (as real users of your API will do). This can

be as simple as providing a few API Notebooks that walk developers through different tasks and then turning them loose on those notebooks to create their own scenarios. Or it can be as complex as creating a written checklist (as is typically used in user experience testing).

If you take the more formal route, it's important to recognize that you will have both concrete sequentials ("I need it in writing, step by step") and abstract randoms ("I want to do this. Oh, and that." "Hey look—a squirrel!"), and you'll want to empower them to utilize their unique personalities and learning/working styles to provide you with a wide scope of feedback.

Your concrete sequential developers will already do things step by step, but your abstract randoms are more likely not to go by the book—and that's okay. Instead of pushing them back onto the scripted testing process, encourage them to try other things (by saying things like, "That's a really interesting use case; I wonder what would happen if…") as again, in real life, this is exactly what developers will do, and this will unlock issues that you never dreamed of.

The purpose of the prototyping process isn't to *validate* that your API is ready for production, but to uncover flaws so that you can *make* your API ready for production. Ideally, in this stage you want to find 99 percent of the design flaws so that your API stands as a solid foundation for future development while also remaining developer-friendly. For that reason it's important not to just test what you've already tested in-house, but to let developers test every aspect of your API. The more transparent your API is, and the more feedback you get from your potential API users, the more likely you are to succeed in this process.

Remember, there's nothing wrong with finding problems. At this point, that *is* the point. Finding issues now lets you circle back to the design phase and fix them before hitting production. Take all of the developers' feedback

to heart—even if you disagree—and watch out for weird things or common themes.

You'll know your API is ready for the real world when you send out the prototype and, after being reviewed by a large group of potential API users (a minimum of 10; 20–50 is ideal), you get back only positive feedback.

6

Authorization & Authentication

Another important aspect of APIs for SaaS providers is authentication, or enabling users to access their accounts via the API. For example, when you visit a site and it says, “Log in with Facebook,” Facebook is actually providing an API endpoint to that site to enable them to verify your identity.

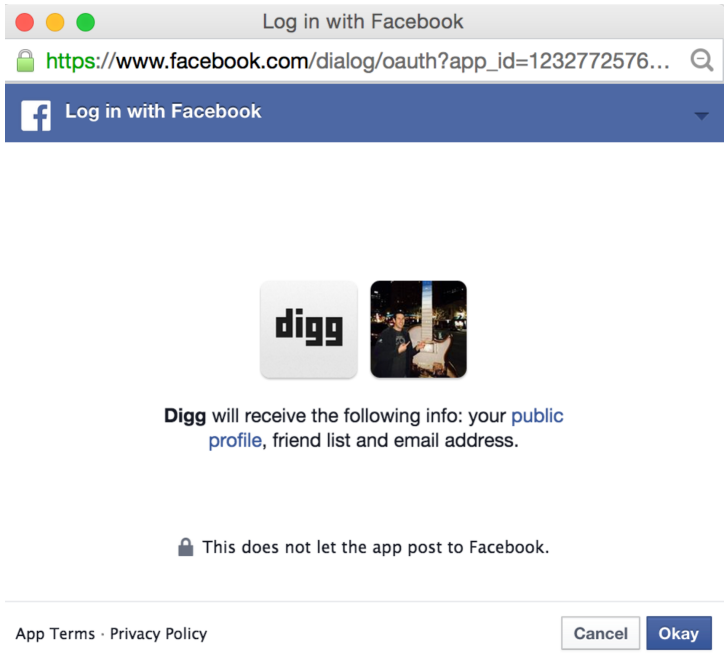
Early on, APIs did this through the use of basic authorization, or asking the user for their username and password, which was then forwarded to the API by the software consuming it. This, however, creates a huge security risk for multiple reasons. The first is that it gives the developers of the software utilizing your API access to your users’ private information and accounts. Even if the developers themselves are trustworthy, if their software is breached or hacked, usernames and passwords would become exposed, letting the hacker maliciously use and access your users’ information and accounts.

To help deal with this issue, Open Authentication—or OAuth—a token-based authorization format was introduced. Unlike basic authorization, OAuth prevents the API client from accessing the users' information. Instead it relays the user to a page on *your* server where they can enter their credentials, and then returns the API client an access token for that user.

The huge benefit here is that the token may be deleted at any time in the event of misuse, a security breach, or even if the user decides they no longer want that service to have access to their account. Access tokens can also be used to restrict permissions, letting the user decide what the application should be able to do with their information/account.

Once again, Facebook is a great example. When you log in to Facebook, a popup comes up telling you that the application wants to access your account and asking you to log in with your Facebook credentials. Once this is done it tells you exactly which permissions the application is requesting, and then lets you decide how it should respond.

This example shows you the Facebook OAuth screen for a user who is already logged in (otherwise it would be asking me to log in), and an application requesting access to the user's public profile, Friends list and email address:



Notice that this is a page on Facebook's server, not on Digg. This means that all the information transmitted will be sent to Facebook, and Facebook will return an identifying token back to Digg. In the event I was prompted to enter a username/password, that information would also be sent to Facebook to generate the appropriate token, keeping my information secure.

Now you may not need as complex as a login as Facebook or Twitter, but the principles are the same. You want to make sure that your API keeps your users' data (usernames and passwords) safe and secure, which means creating a layer of separation between their information and the client. You should never request login credentials through public APIs, as doing so makes the user's information vulnerable.

Generating Tokens

It's also extremely important to ensure that each token is unique, based both on the user and the application that it is associated with. Even when role-based permissions are not required for the application, you still do not want a generic access token for that user, since you want to give the user the ability to have control over which applications have access to their account. This also provides an accountability layer that allows you to use the access tokens to monitor what an application is doing and watch out for malicious behaviors in the event that they are hacked.

It's also smart to add an expiration date to the token, although for most applications this expiration date should be a number of days, not minutes. In the case of sensitive data (credit cards, online banking, etc) it makes more sense to have a very short time window during which the token can be used,, but for other applications, doing so only inconveniences the user by requiring them to login again and again. Most access tokens last between 30 and 90 days, but you should decide the timeframe that works for you.

By having the tokens automatically expire, you are adding another layer of security in the event that the user forgets to manually remove the token and are also helping to limit the number of applications that have access to your users' data. In the event that the user wants that application to be able to access their account, they would simply reauthorize the app by logging in through the OAuth panel again.

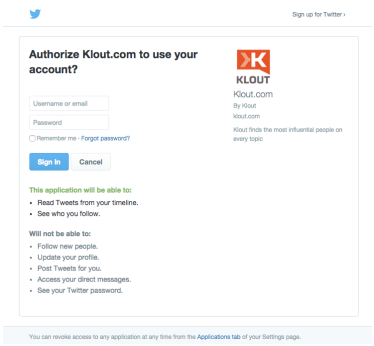
Types of OAuth

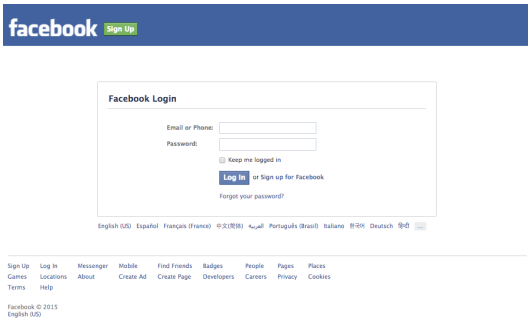
Unfortunately, implementing OAuth may not be as simple as one would hope, as there are two very distinct versions—OAuth 1 and OAuth 2—with OAuth 2 having several different possible implementations. Many companies also choose to deviate from standards and create their own style of OAuth 2 instead.

In general, OAuth 2 is considered to be less secure than OAuth 1, as it relies on SSL instead of signed certificates. But what it may lack in security, it makes up for in flexibility and usability. Because OAuth 2 does not require custom-signed certificates for communication, but instead relies on SSL, it is easier for third-party developers to implement. OAuth 2 also reduces the number of tokens needed from two down to one—a single token that is designed to be short-lived and easily removed either through timed expiration or by the user at any point. Lastly, OAuth 2 offers a unique layer of flexibility by letting the user choose which actions the calling application has access to, which limits what third-party companies can do with their data and account.

When choosing a type of OAuth, it's important to understand the key differences between the two, and how choosing one or the other might impact usability and security. Regardless of whether you choose OAuth 1 or OAuth 2, you're still choosing a much better route than giving calling applications access to your user's credentials.

In the rest of this chapter we'll be focusing on OAuth2, as this offers the most flexibility and may be the best choice for most APIs. But this is a question that is best answered by security experts who can truly understand what you are trying to accomplish with your API. It's also important to understand that both of these formats are still heavily relied on today, with Twitter utilizing OAuth 1.0A and Facebook taking advantage of a slightly customized version of OAuth 2.

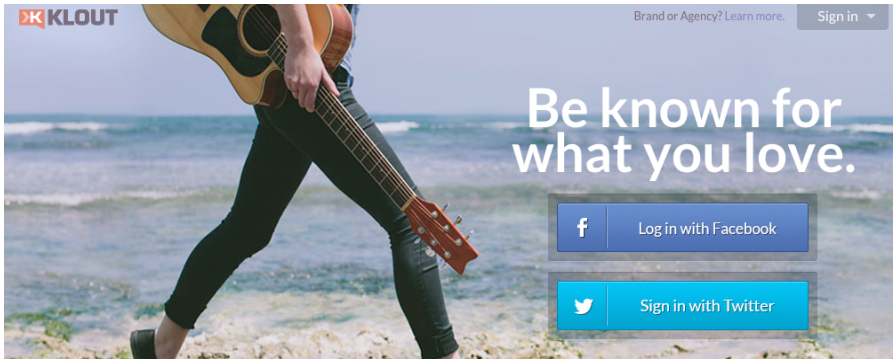




Twitter expands on OAuth 1 by restricting access to protected resources ("This application will be able to..., but will not be able to…). This in itself is not a feature of OAuth 1, but shows that even when using OAuth Version 1.0, an application can expand upon it to limit what a token can access. See http://oauth.net/core/1.0/#anchor42 for more information on.

OAuth2

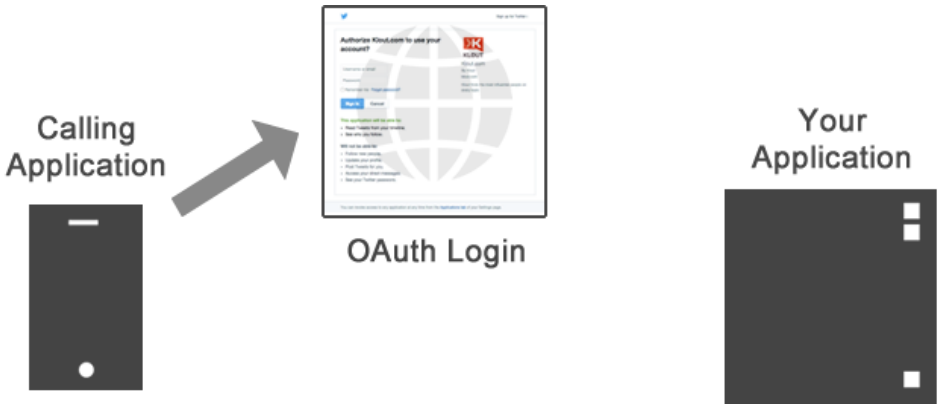
In a two-legged OAuth 2 process, the application consuming your API first prompts the user to log in using your service. This is usually done through the use of a "log in with" button, as shown on the Klout website:



Within this button or link is crucial information to complete the hand shake, including what the application is requesting (a token), the URI to which your

application should respond (such as http://theirsite.com/oauth.php), the scope or permissions being requested and a client ID or unique identifier that allows their application to associate your response with that user.

Now, when the user clicks the button (or link), they are then redirected to *your* website (with all the above information being transmitted), where they are able to log in and determine which permissions they want the application to have access to (as shown in the OAuth screenshots for Twitter and Facebook above).



Your application then generates an access token based on both the user and the application requesting access. In other words, the access token is tightly coupled to both the user and the application, and is unique for this combination. However, the access token can be independent of access permissions or scope, as you may choose to let the user dictate (or change) these permissions from within your application. By having the scope remain changeable or decoupled from the hash of the token, users are able to have any changes they make regarding the scope from within your application applied immediately without needing to delete or regenerate a new token.

The access token created should also have a set expiration (again, usually days, but this should depend on your API's needs). This is an additional

security measure that helps protect a user's information by requiring them to occasionally reauthorize the application requesting access to act on their behalf. (This is often as simple as clicking "reauthorize" or "login with….")



Once the access token has been generated, your application then responds back to the URI provided by the application to provide the unique identifier or client ID and access token that the application may utilize to perform actions or request information on their behalf.

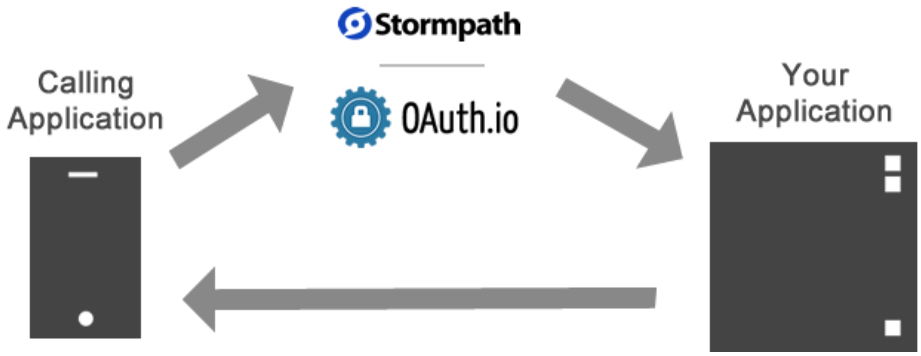


Because this information is not being handled through signed certificates, it is important that the information being transmitted is handled over SSL. However, to be truly secure, BOTH parties must implement SSL. This is something to be aware of as many API users may not realize this and instead create insecure callback URLs such as "http://theirdomain.com/oauth.php" instead of "https://theirdomain.com/oauth.php." For this reason, you may want to build in validation to ensure that you are passing data back to a secured URL in order to prevent interception of the access token by malicious third parties.

As an added security measure, you can also restrict the access token to the domain of the calling application.

Once the application receives the access token and client ID or identifier, it can then store this information in its system, and the handshake is complete until the access token either expires or is deleted by the user. At that time, should the user choose to reauthorize the application, the handshake starts back at the beginning.

In a three-legged OAuth process, the flow is the same, with the exception of having one more party involved (such as an OAuth service provider) who would then act as the middle leg and provide your application with the information.



OAuth and Security

When implementing OAuth it's important to understand that it is the only thing preventing free access to your users' accounts by the application—and any malicious users who try to hijack or abuse it.

This means that you need to take a security-first approach when building out your OAuth interface, and that before building anything on your own it is important to first understand your own security needs, and secondly understand the different security aspects (and vulnerabilities) of OAuth.

Brute Force and Man-in-the-Middle Attacks

Attackers may attempt to use brute force attacks against your OAuth solution or utilize a man-in-the middle attack (pretending to be your server and sneaking into the calling application's system that way).

Improper Storage of Tokens

It's also important to remember that your users' information is only as secure as their access tokens. I've already mentioned being sure to make all calls over SSL, but you should also work with your API users to ensure they are properly and securely storing access tokens.

Many years ago while working on a shopping cart system for a client, I came across a file freely available in the root called "config.xml." This config file contained their database username and password, where the user was not restricted by IP or host. In other words, anyone who found that file had access to everything in their database! When I talked to the client about it, they shrugged it off, saying, "Well who would ever look for that?"

Unfortunately, that is not the only frightening security tale I have. I've also seen access tokens being passed over JavaScript—freely visible in HTML code—and on non-secure pages.

The point is that just because some things are common sense to you, you should not assume the same of your API users—especially when it involves your users' data. Be sure to educate them and guide them along the way to help ensure these vital access tokens remain secure and out of the public eye.

Session Fixation/Hijacking

If you apply the principles of REST to your OAuth solution and keep it stateless, you should be able to avoid many of the security risks associated with session hijacking and fixation. However, this becomes a real threat under two circumstances—first when the issue is on the API user's side, and the second when the OAuth screen is being accessed via a public or shared computer.

Unfortunately there is not much you as the provider can do in the first place other than watch for multiple access token requests for the same user. (If you start seeing several requests per second for the same user or client ID/call identifier, chances are something isn't right —it could be a loop in their system or an indication that they have a session/client ID issue).

In the second case, many applications such as Facebook automatically jump to the permissions screen instead of forcing the user to log in—if they are already logged into their platform. While this is incredibly convenient for the user, it may lead to unintended or unauthorized access of their accounts by third parties if they forget to log out. This is something you should consider when determining how to handle OAuth sessions on your side.

Security Isn't Easy

As you can see, security in regards to OAuth isn't just about what you build or the code you write. It extends into how information is stored and

communicated, since the ability to access user information and accounts is priceless to hackers with malicious intent. This makes your API—especially as your application grows—a prime source for such attacks.
One of the biggest mistakes developers make when dealing with security is trying to secure systems without truly understanding the different security measures necessary. Perhaps the best talk I have heard on encryption, by Anthony Ferrara, ended with this piece of advice: “Don’t. Hire a professional.”

The same goes with implementing OAuth. Unless you are comfortable with signed certificates, RSA tokens, HMAC, consumer/token secrets, SHA1, etc.—and I mean REALLY comfortable—you may want to either look at bringing in someone with experience to help you build this out instead, or take advantage of a common OAuth library or an OAuth service provider such as OAuth.io or StormPath.

Adding OAuth to RAML

The good news is that once you have an OAuth service, adding it to your API’s definition in RAML, and making it accessible through the different tools available, is extremely easy.

For OAuth 1, you would simply need to state that it is securedBy oauth_1_0 and provide a requestTokenUri, an authorizationUri and the tokenCredentialsUri as shown below in the Twitter RAML example:

```
  securitySchemes:
    - oauth_1_0:
       type: OAuth 1.0
       settings:
         requestTokenUri:
https://api.twitter.com/oauth/request_token
         authorizationUri:
https://api.twitter.com/oauth/authorize
```

```
        tokenCredentialsUri:
https://api.twitter.com/oauth/access_token
  securedBy: [ oauth_1_0 ]
```

For OAuth 2, you would likewise state that it is securedBy oauth_2_0 and provide the accessTokenUri and authorizationUri. Because OAuth 2 only uses one token, we are able to combine the requestTokenUri and tokenCredentialsUri URIs into the same request (accessTokenUri). However, because OAuth 2 utilizes scope, we will need to add that in using the scope's property. We'll also need to add the information on how to send the access token via the Authorization header:

```
securitySchemes:
  - oauth_2_0:
type: OAuth 2.0
      describedBy:
        headers:
          Authorization:
            description: |
              Used to send valid access token
            type: string
settings:
        authorizationUri:
https://api.instagram.com/oauth/authorize
        accessTokenUri:
https://api.instagram.com/oauth/access_token
        authorizationGrants: [ code, token ]
        scopes:
          - basic
          - comments
          - relationships
          - likes
securedBy: [ oauth_2_0 ]
```

You can learn more about using OAuth within RAML in the RAML spec under “Security” at http://raml.org/spec.html#security. But thankfully the process of implementing existing OAuth services into your RAML-based applications is far more simple than actually creating them, and it makes it easy for your developers to access real information when debugging or exploring your API.

7

Designing Your Resources

Resources are the primary way the client interacts with your API, and as such it's important to carefully adhere to best practices when designing them, not only for usability purposes, but to also ensure that your API is long-lived.

In REST, resources represent either object types within your application or primary gateways to areas of your application. For example, the /users resource would be used to interact with and modify user data. In a CRM application you may have a /users resource that would represent the users of the application, and a /clients resource that would represent all of the clients in the application. You might also have a /vendors resource to manage suppliers, /employees to manage company employees, /tickets to manage customer or sales tickets, etc. Each resource ties back into a specific section of the CRM, providing a general gateway to reach and interact with that specific resource.

But what makes REST truly unique is that the resources are designed to be decoupled from their actions, meaning that you can perform multiple actions on a single resource. This means that you would be able to create, edit and delete users all within the /users resource.

Decoupled Architecture

If we look at the constraints of REST, we have to remember that REST is designed to be a layered system that provides a uniform interface. It must also allow the server to evolve separately from the client and vice versa.

For this reason, resources are designed to be decoupled from your architecture and tied not to specific methods or classes, but rather to generalized application objects. This is a big change from SOAP, where the calls are tied to the class methods, and RPC, where naming conventions tend to be tightly coupled to the action you're taking (getUsers).

By decoupling your resources from your architecture, you are ensuring that you can change backend services or technologies without impacting how the client interacts with your API while also providing flexibility in how the client interacts with your API through the use of explicit methods or representations.

Use Nouns

One of the best ways to ensure that your resources are decoupled is to think of them as webpage URIs. For example, if sharing information about your company, you would probably send the user to an "about" section on your website. This might look something like "yourdomain.com/about" or "yourdomain.com/company."

In the same way, you can build out resources using that same navigational principle. As mentioned above in the CRM example, users could be

directed to the /users resource, clients to the /clients resource, vendors to /vendors, etc.

Another way to be sure that you are enforcing this navigational style and avoiding tight coupling of your resources to methods or actions is to utilize nouns for the name of your resource. If you find yourself using verbs instead, such as "/getUsers" or "/createVendor," there's a good chance you're creating a tightly coupled resource that is designed to only perform one or two actions.

Resources should also take advantage of the plural form. For example, /users represents the full scope of the user object, allowing interaction with both a collection (multiple records) and an item (a single user). This means that the only time you would want to take advantage of a singular noun is if the only possible action that can be taken is specific to a single item or entity. For example, if you were creating a shopping cart API, you may elect to utilize the singular form "/cart" for the resource rather than "/carts." But again, in general, the plural form will offer you the most flexibility and extendibility, as even when a feature is not built into your application, or there are current restrictions (for example, only letting companies have one location) that may change in the future. And the last thing you want is to have both a plural and singular form of the resource. For example, imagine users having to decide whether to use /location or /locations.

In other words, only use the singular format when there's no possibility of the resource having multiples—a scenario that is extremely rare. After all, even in the /cart example, you may decide someday to give users multiple shopping carts (saved carts, business carts, personal carts, etc.). So as you build out your resources, you should be thinking not just about planning for now, but planning for what could happen down the road as well.

Content-types

Resources should also be able to support multiple content-types, or representations of that resource. One of the most common mistakes developers are making today is building their API for only one content-type. For example, they will building out their REST API and only return JSON in the response. This by itself is not necessarily a bad thing, except that they are failing to inform their clients that other media types are not allowed (see status codes and errors), and even more importantly, they have not built out their architecture to be flexible enough to allow for multiple content-types.

This is very shortsighted, as we forget that only a few years ago XML was king. And now, just a short while later, it is repeatedly mocked by "progressive developers," and the world is demanding JSON (and for good reason).

With the emergence of JSON, many enterprises were caught off guard, stuck serving XML via SOAP APIs with no new way to meet their customers' needs. It is only now that we are seeing many enterprises in a position to provide RESTful APIs that serve JSON to their customers.

The last thing we want to do is put ourselves in this position again. And with new specs emerging every day, it is just a matter of time. For example, YAML (Yet Another Markup Language) is already gaining popularity, and while it may not be the primary choice for most developers today, that doesn't mean some of your most influential clients won't ask you for it.

By preparing for these types of scenarios, you also put yourself in a position to meet all of your clients' needs and provide an extremely flexible and usable API. By letting developers decide which type of content-type they are utilizing, you let them quickly and easily implement your API in their current architecture with formats they are comfortable with. Surprisingly, along with functionality and flexibility, this is something that many developers are looking for.

Using the Content-type Header

Today, when you browse the Web your client (browser) sends a content-type header to the server with each data-sending request, telling the server which type of data it is receiving from the client.

This same principle can be applied to our HTTP based REST API. For example, you can use the content-type header to let clients tell your API what format of data they are sending you, such as: XML (text/xml, application/xml), JSON (application/JSON), YAML (application/yaml) or any other format that you support.

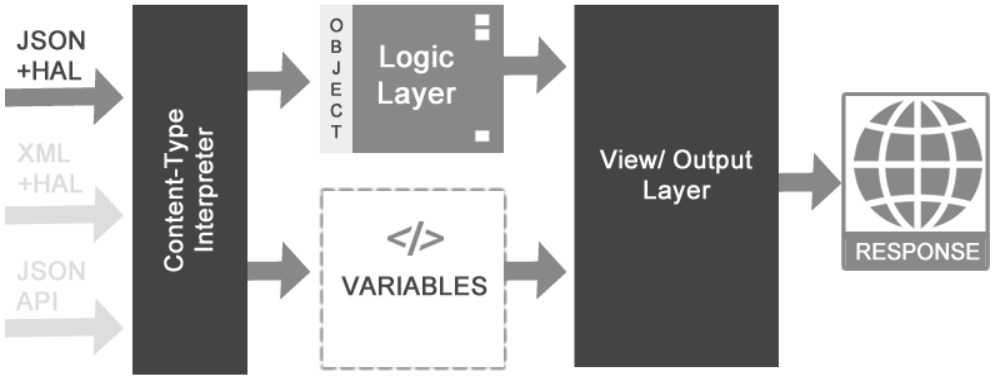
Once the server receives this content-type, it not only knows what data it has received but, if it is a recognized format, how to process it as well. This same principle can be applied to your API, letting you know which type of data your client is sending and how to consume it. It also tells you which data format they are working with.

To go a step further, you can also take a look at the Accept header to see which type of data they are expecting in return. Hypothetically, when you build out your architecture your client should be able to send you XML by declaring it in the content-type and expect JSON in return by declaring a desired JSON response in the Accept header.

This creates the most flexibility for your API and lets it act as a mediator when used in conjunction with other APIs. However, it also provides a lot of opportunity for confusion. Because your API is designed to have a uniform interface, I would recommend not taking advantage of this wonderful header, but rather relying on the content-type to determine which data format they are working with, and then passing back that same format.

Building Your Architecture

In order to best accommodate different content-types, you will want to build out two specific areas of your API's architecture—an incoming content handler and an outbound view renderer.

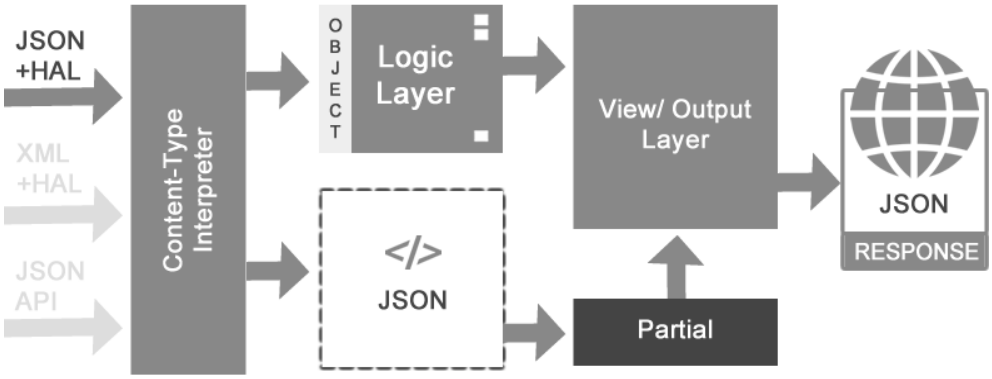


Unless you have to create special schemas for the data, it's recommended to keep the data architecture the same regardless of which format you are using. This lets you easily break down the incoming data using an incoming content-type handler, formatting it quickly into a usable object that can easily be interpreted by your API service layer and passed on to the necessary services to generate the data necessary for a response.

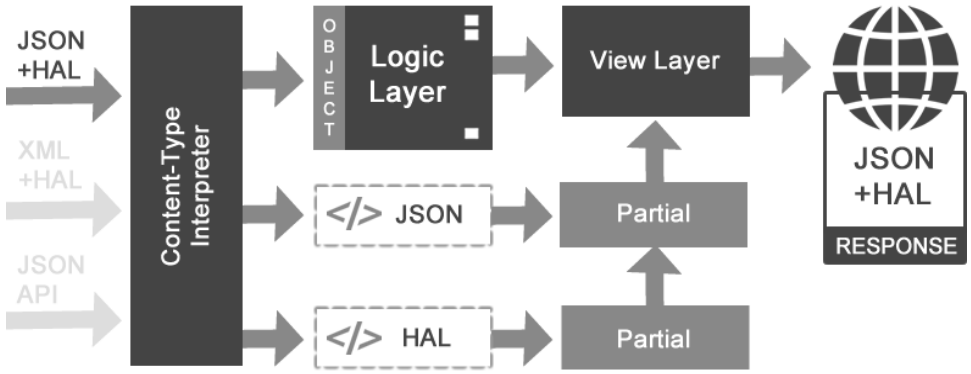
The problem with having specialized architectures or schemas is that you will need a custom class to handle the deserialization of your incoming data and assign it to the appropriate class/property architecture so that it can be processed by your underlying service architecture. While this can quickly be done, unless it is carefully planned out, you may find yourself having to build this interpretation layer for each resource and add new components to it each time a new component is added.

By having a standardized architecture, or having your data organized in the same manner across multiple formats (XML, JSON, YAML, etc.), you will be able to reduce the amount of work needed to provide this layer of flexibility. This is not always possible, however, and tools like MuleSoft's Anypoint Studio with Mule ESB may make the task easier through the use of their DataMapper.

In the case of output, you'll want to take advantage of a view renderer. This lets you build your architecture in such a way that the response format is separate from the data processing and generation. Again, if you have specialized schemas, you will have to write special classes/objects to handle them, but if you have a standardized response architecture, you should be able to simply pass the response data to the view layer based on the content-type used by the client.



When building your view renderer, you may want to create a separate view layer for handling hypertext links, which will let you generate the appropriately formatted content response and then pull in the hypermedia link representations formatted in the requested content-type (HAL, JSON API, CPHL, Siren, etc.).



Many of today's popular MVC frameworks already take advantage of view renderers and partials (incomplete view aspects such as those that could be used to generate the hypertext links), making this process extremely simple.

But in the event that you have to build it out, remember by creating your content-handler and content-response views in a layered format, you will be able to quickly add new response types with little to no work on your part. For some frameworks with a standardized data architecture, this is as easy as adding a new view renderer for the content-type you want to support, and then adding that content-type to the list of accepted types. Of course, you still have to build the content-handler to deserialize the data, but with the layered system, supporting a new content-type may only take a matter of hours or days (not including testing) instead of taking months, years or a major rewrite. This will allow you to quickly adapt and support your clients' needs as technology evolves.

XML

Defined by the W3C, XML or the Extensible Markup Language was designed to present a format that was both machine- and human-readable. Some of the more common formats of XML include RSS (commonly used for feeds), Atom, XHTML and, of course, SOAP.

XML also encourages the use of strict schemas and was the choice format for many enterprises, causing the move to JSON to be more challenging.

However, while descriptive, XML takes up more bandwidth than JSON, and while commonly used, does not have the same broad language support. While there are many libraries for interpreting XML, many of these are used as add-ons rather than core libraries.

```
<books>
   <book>
      <title>This is the Title</title>
      <author>Imag E. Nary</author>
      <description>
         <![CDATA[Once upon a time there was a great book]]
      </description>
      <price>12.99</price>
   </book>
   <book>
      <title>Another Book</title>
      <author>Imag E. Nary</author>
      <description>
         <![CDATA[This is the sequel to my other book]]
      </description>
      <price>15.99</price>
   </book>
</books>
```

JSON

JSON, or the JavaScript Object Notation, was designed as an alternative to XML, consisting of key/value pairs in a human-readable format. Originally created by Douglas Crockford for use within JavaScript, JSON is now a language agnostic, being described in two separate RFCs, and has quickly

become one of the most commonly used formats due to its broad language support and ability to serialize objects.

JSON is represented by the application/json content-type and typically takes advantage of the .json extension.

```
{[
   {
      "title" : "This is the Title",
      "author" : "Imag E. Nary ",
      "description" : "Once upon a time there was a great
      book ",
      "price" : "12.99",
   },
   {
      "title" : "Another Book",
      "author" : "Imag E. Nary ",
      "description" : "This is the sequel to my other
      book",
      "price" : "15.99",
   },

]}
```

You can also define strict JSON through the use of JSON Schemas, although these are not as commonly used.

YAML

YAML, or Yet Another Markup Language/YAML Ain't Markup Language (depending on if you go by the original meaning or the latter acronym), was originally created by Clark Evans, Ingy döt Net and Oren Ben-Kiki with the goal of being a simpler, more human-readable format.

To accomplish this goal, YAML utilizes whitespace to identify properties, eliminating the need for opening/closing brackets in most instances. However, support for YAML has been slow, with many languages lacking core libraries for its serialization and deserialization. This may be due in part to the complex rules YAML incorporates, providing users with useful shortcuts in building their files out, but making the actual deserialization process much more difficult.

While YAML hasn't been widely adopted as a response format, it has become the format of choice for API definition and modeling languages, including both RAML and Swagger.

```
Books:
   -
      title: This is the Title
      author: Imag E. Nary
      description: |
         Once upon a time there was a great book
      price: 12.99
   -
      title: Another Book
      author: Imag E. Nary
      description: |
         This is the sequel to my other book
      price: 15.99
```

Versioning

I cannot stress enough that when it comes to building an API, your goal should be to create one that is so amazing that you can avoid versioning altogether. However, as hard as we try, there is a good chance (at least with today's technology) that at some point in time we will find ourselves having to version our API for one reason or another.

There have been several suggested methods for versioning, but the first thing to remember is that, fundamentally, versioning is a bad thing. It's important to understand that the lower the version number, the better. In the desktop software world, we push for that next number, creating Version 1, 2 and—in some cases—skipping numbers altogether to make the product sound more advanced than it is! But in the API world, the sign of a success is not having to version.

This means you should consider avoiding minor versioning altogether, since it serves no real purpose in an API. Any feature changes should be made immediately and seamlessly available to developers for their consumption. The only strong argument for minor versioning is in tightly coupled SDKs, or saying this SDK supports the features of Version 1.1, whereas access to recently added features would require an upgrade to Version 1.2.

You could also make the argument that minor versioning lets developers quickly know there's been a change to your API—an argument that makes sense on the surface. Of course the counter argument is that you may have developers who misunderstand minor versioning and instead rush to try and upgrade their system to the new API without needing any of the new features (or while they're already taking advantage of them without realizing it). This may result in unnecessary support calls and emails, as well as confusion ("Can I do this in 1.1 or only in 1.2? And how do I access version 1.1 instead of 1.2?").

The other counterpoint to this argument is that if you build a strong developer community, developers will talk about new features (although not everyone will be involved in the community), and if you utilize registration to gain an API key (spoiler alert— you should) you can keep in touch with all of your developers via email. (They may not be read, but then again, minor versioning in the code might not be seen either.)

So with this in mind, let's take a look at the three different mainstream schools of thought regarding how to version your API.

In the URI

This method includes the version number in the base URI used for API calls, making developers explicitly call the API version that they want. For example, to access Version 1 of the API, one would use api.domain.com/v1/resource to access the API, whereas for Version 2 they would call api.domain.com/v2/resource. This means that when reading documentation and implementing your API, developers will be forced to look at the version number, since they may not notice it when briefly looking over code samples unless it is predominately called out. This makes this method preferable for APIs that are catering to newer developers.

One argument against the URI method is that it doesn't allow the API to be hypermedia driven, or that the content-type method makes this easier. This is partially because REST is designed to be hypermedia driven and not tightly coupled to URIs, which URI versioning does. Also, most hypermedia specs rely on relative path text links that utilize the base URI, meaning that unless there is an explicit change made by the developer, the client will always call the same version of the API, staying within the current realm and not being able to move back and forth between versions automagically.

However, even with the content-type, we currently have no good way to know what the client supports, So when calling Version 2 from a client that only supports certain Version 2 segments, we're still likely to get back Version 2 links in the hypertext response, causing the client application to fail.

In other words, the problem cannot be avoided regardless of whether you're using the URI or the content-type to denote the version. Although as applications become more advanced with machine learning and code-on-demand, we may see this change, but I feel that this can be accommodated regardless of which method you are using, just perhaps not as cleanly as with the content-type versioning method.

One advantage of URI versioning is that it tells developers which version they are using, and is easier for newer developers to implement. It also helps prevent confusion if developers forget to append the version on top of the content-type version type (which if using this method should throw an error to prevent ambiguity).

Of course, it's also very easy for the base URI to become hidden somewhere in an include, meaning that developers may not explicitly know which version of the API they are using without having to dig into their own code. Just the same, the other methods run this same risk depending on how the client's application is architected.

In the Content-type Header

This method is arguably cleaner and far less coupled than the URI method. With this method, developers would append the version to the content-type, for example:

```
Content-type: application/json+v1
```

This lets developers quickly modify the version for the calls needed and reinforces the use of representations to communicate with the server. It also allows for a more dynamic and hypermedia-led API, as one could implement the Accept header to return back a specific version. (For example, if I make a call on a V2 feature, I may ask to only return V1 links in the response, as other applications of my API may not be compatible). Although doing this could create an architectural nightmare (What happens if you do a create? Do you only return V2 responses for that item, and then V1 for resources that have a shared relationship? What about compatibility issues?).

This also raises questions regarding a uniform interface, as you are transitioning the user between two incompatible versions of your API to accomplish different things. On the other hand, this may help developers

transition from one version to another, as they can do it over time instead of all at once. Just the same, I can't say it is recommended, as I believe that depending on business needs and implementation, it may cause far more harm than good .

Another issue with the content-type is that developers have to *know* that they need to call this out. This means that you have to not only have clear documentation, but also validation regarding whether or not this information is provided.

You must also have a central routing mechanism between your two APIs, which presents a possible domain challenge. Since a key reason you are versioning is that your current version no longer meets your needs, you are probably not just rebuilding one section of the API, but rather its very foundation. This may make taking advantage of the content-type method of versioning far more complex than having multiple, but explicit, URIs.

Perhaps the biggest benefit of the content-type method is if you have two different versions of your application (some customers are on V1, some on V2) and you want to provide an API that can accommodate both. In that case you're not really versioning your API, but rather letting customers tell you which version of your application they're on so you can provide them with the appropriate data structures and links. This is an area where the content-type method absolutely excels.

Outside of this use case, content-type method falls prey to many of the same problems as the URI method, in addition to creating more work and opening you up to "out of the box" use cases (such as people trying to take advantage of the Accept header in conjunction with the content-type header to go between versions).

In a Custom Header

The custom header is very similar to the content-type header, with the exception that those using this method do not believe that the version belongs in the content-type header, and instead makes sense in a custom header, such as one called “Version.”

```
Version: 1
```

This helps prevent the whole “Accept” conundrum, but it also runs into the same issues of the content-type header as well as forces developers to veer into the documentation to understand what your custom header is, eliminating any chance of standardization (unless everyone decides on a standard custom header, such as “version”).

This also opens up confusion, as developers may ask how to send a version number through a header and get multiple answers ranging from other API’s custom headers to using the content-type header.

For that reason, I cannot recommend using the custom header. And while I personally agree that the content-type header may not be the best place either, I think using a pre-existing, standard header is better than creating an offshoot—at least until a new header is established and standardized for this purpose.

Caching

One of the primary constraints of REST, caching is critical to scaling your API. As you build out your API’s application and services layer, you will of course want to take advantage of caching mechanisms within your own code. But while this will help reduce the data load and memory usage, letting developers cache their calls by providing them with the necessary information eliminates unnecessary calls. This of course lets your API do more and helps protect you against memory leaks (hopefully letting you find

them before they crash your server), while also allowing you to scale without increasing cost.

However, before developers can cache their calls, they need to know that the call *is* cacheable and how often they should renew their cache when the data for that call expires. Surprisingly, this is one of the most forgotten aspects of REST style APIs.

You can share this information with developers through the headers you send, including the Cache-Control and Expires header. In the Cache-Control header you can declare whether or not the cache should be public or private (used with CDNs) or whether or not the data should not be cached (no-cache) or stored (no-store). You can also add a max-age (in seconds) for which that the data should be stored and a number that is designed to override the Expires header (but this is reliant on client implementation).

Because you can use either the Expires or the max-age, your cache headers might look like:

```
Cache-Control: public, max-age=3600
```

With this example we are declaring that the cache is public (meaning it can be cached by both a CDN or the client), and that the cache should expire in 1 hour (or 3600 seconds). Using the Expires header instead, your headers would look like this:

```
Cache-Control: public
Expires: Mon, 09 February 2015 17:00:00 GMT
```

In this example, we are still sending back data that is publicly cacheable but expires explicitly on February 9, 2015 at 5 p.m. GMT. In the event we did not want the developer caching the response (for when they need real

time data) we could instead send back the cache-control header with the no-cache (do not cache the data) and no-store (do not store the data) declarations.

```
Cache-Control: no-cache, no-store
```

However, telling developers that your API is cacheable in the headers alone is not enough. You'll also want to inform them in the documentation, letting them know which resources are cacheable and encouraging them to do so.

If you provide an SDK or a code wrapper, you will also want to implement caching into your code to help developers who are looking for a plug-and-play solution.

8

Designing Your Methods

Similar to having a class with methods for performing different actions on an object, REST utilizes methods within the resources. For a web-based REST API that will be accessed over the HTTP or HTTPS, protocol we can take advantage of the predefined, standardized HTTP methods. These methods represent specific actions that can then be tied to the CRUD acronym.

Utilizing CRUD

CRUD stands for Create, Read, Update and Delete and is an acronym commonly used when referring to database actions. Because databases are data-driven, like the Web, we can apply these same principles to our API and how our clients will interact with the methods.

This means that we will be utilizing specific methods when we want to create new objects within the resource, specific methods for when we want

to update objects, a specific method for reading data and a specific resource for deleting objects.

However, before we can apply CRUD to our API, we must first understand the difference between interacting with items verses collections, as well as how each method affects each one. This is because multiple methods can be used for both creating and updating, but each method should only be used in specific cases.

Items versus Collections

Before we continue, we need to define the difference between an item and a collection. An item is a single result set, or a single data object, such as a specific user. A collection, on the other hand, is a dataset comprised of multiple objects or multiple users. This differentiation is important because while some methods are appropriate for collections, other methods are appropriate for dealing with an item.

For example, when dealing with a collection you have to be very careful when allowing updates or deletes, as an update on a collection will modify every record within it, and likewise a delete will erase every single record. This means that if a client accidentally made a delete call on a collection, they would effectively (and accidentally) delete every single user from the system.

For this reason, if you plan to let your users do mass edits/ deletes, it's always a good idea to require an additional token in the body to ensure that they are doing exactly what they are intending. Remember, REST is stateless, so you should not force them to make multiple calls, as you have no way of carrying over state on the server side.

For single, pre-existing records, it makes perfect sense to let a user edit or even delete the record. However it doesn't make much sense to let them

create a new record from within a specific record. Instead, creation should be reserved for use on the collection.

While this can be a little confusing at first, with proper documentation and the use of the OPTIONS method, your API users will be able to quickly identify which methods are available to them. As they work with your API, this will eventually become second nature as long as you remain consistent in their usage.

HTTP Methods

You're probably already quite familiar with HTTP methods, or HTTP action verbs. In fact, every time you browse the Web with your browser, you are taking advantage of the different methods—GET when you are requesting a website and POST when you are submitting a form.

Each HTTP method is designed to tell the server what type of action you want to take, ranging from requesting data, to creating data, to modifying data, to deleting data, to finding out which method options are available for that given collection or item.

Of the six methods we're going to look at, five can be mapped back to CRUD. POST is traditionally used to create a new object within a collection, GET is used to request data in a read format, PUT and PATCH are used primarily for editing existing data and DELETE is used to delete an object.

However, there is some crossover among the different methods. For example, while POST is predominately used to create objects, PUT can also be used to create an object within a resource—if it doesn't already exist. Likewise, PUT and PATCH can both be used to edit existing data, but with very different results. For that reason it's important that you understand what each method is intended to do, and which ones you should use for what purpose. This is also something you'll want to explain

in your documentation, as many developers today struggle with understanding how to use PUT to create, as well as the difference between PUT and PATCH.

GET

The GET HTTP Method is designed explicitly for getting data back from a resource. This is the most commonly used HTTP Method when making calls to a webpage, as you are getting back the result set from the server without manipulating it in any way.

In general, a GET response returns a status code 200 (or ok) unless an error occurs, and relies on a querystring (domain.com/**?page=1**) to pass data to the server.

```
HTTP/1.1 200 OK
cache-control: public, max-age: 3600
Content-Type: application/json
Date: Tue, 31 Mar 2015 02:40:01 GMT
Vary: Accept-Encoding
Content-Length: 72
Connection: keep-alive
```

The GET method should be used any time the client wants to retrieve information from the server without manipulating that data first.

POST

One of the most versatile HTTP Methods, POST was designed to create or manipulate data and is used commonly in Web forms. Unlike GET, POST relies on body or form data being transmitted and not on the query string. As such, you should not rely on the query string to transmit POST data, but rather send your data through form or body data, such as in JSON.

While extremely versatile, and used across the Web to perform many different functions due to its common acceptance across multiple servers, because we need an explicit way to define what type of action should be taken within a resource, it is best to only use the POST method for the creation of an item within a collection or a result set (as with a multi-filtered search).

When creating an object (the function for which POST should predominately be used), you will want to return a 201 status code, or Created, as well as the URI of the created object for the client to reference.

```
HTTP/1.1 201 Created
cache-control: public, max-age: 3600
Content-Type: application/json
Date: Sun, 12 Apr 2015 21:07:13 GMT
Location: http://api.domain.ext/resource/207
Vary: Accept-Encoding
Content-Length: 72
Connection: keep-alive
```

PUT

Less well known is the PUT Method, which is designed to update a resource (although it *can* create the resource if it doesn't exist).

Traditionally, PUT is used to explicitly edit an item, overwriting the object with the incoming object. When using the PUT method, most developers are not expecting an object to be created if it doesn't exist, so taking advantage of this clause within this method should be done with extreme care to ensure that developers know exactly how your API uses it. It's also important that your usage of PUT remains consistent across all resources. (If it creates an object on one resource, it should do the same on all the others.)

If you elect to utilize PUT to create an item that doesn't exist (for example, calling "/users/1" would create a user with the ID of 1), it is important to return the Created status code, or 201. This tells your consumers that a new object was created that may (or may not) have been intended.

It's also important to understand that you cannot use PUT to create within the resource itself. For example, trying a PUT on /users without explicitly stating the user ID would be a violation of the standardized specification for this spec.

For this reason I would highly recommend *not* creating an object with PUT, but rather returning an error informing the client that the object does not exist and letting them opt to create it using a POST if that was indeed their intention. In this case, your request would simply return status code 200 (okay) if the data was successfully modified, or 304 if the data was the same in the call as it was on the server.

```
HTTP/1.1 304 Not Modified
cache-control: public, no-store, must-revalidate
Date: Tue, 31 Mar 2015 02:48:23 GMT
Connection: keep-alive
```

It's also important to explain to your users that PUT doesn't just overwrite the object data that they submit, but *all* of the object data. For example, if I have a user with the following structure:

```
{"firstName" : "Mike",
 "lastName" : "Stowe",
 "city" : "San Francisco",
 "state" : "CA"}
```

And I submit the following request using a PUT:

```
{"city" : "Oakland"}
```

The object on the server would be updated as such, reflecting a complete override:

```
{"firstName" : "",
 "lastName" : "",
 "city" : "Oakland",
 "state" : ""}
```

Of course, this is traditionally not what the user wants to do, but is the effect of PUT when used in this case. What the client should do when needing to patch a portion of the object is to make that same request using PATCH.

PUT should never be used to do a “partial” update.

PATCH

Another lesser-known method, PATCH has created some confusion, as it operates differently as an HTTP method than when used in bash/shell commands.

In HTTP, PATCH is designed to update only the object properties that have been provided in the call while leaving the other object properties intact.

Using the same example we did for PUT, with PATCH we would see the following request:

```
{"city" : "Oakland"}
```

Which would return the following data result set from the server:

```
{"firstName" : "Mike",
 "lastName" : "Stowe",
 "city" : "Oakland",
 "state" : "CA"}
```

Like PUT, a PATCH request would return either a 200 for a successful update or a 304 if the data submitted matched the data already on record—meaning nothing had changed.

```
HTTP/1.1 200 OK
cache-control: public, max-age: 3600
Content-Type: application/json
Date: Tue, 31 Mar 2015 02:49:47 GMT
Vary: Accept-Encoding
Content-Length: 72
Connection: keep-alive
```

DELETE

The DELETE Method is fairly straight forward, but also one of the most dangerous methods out there. Like the PUT and PATCH methods, accidental use of the DELETE method can wreck havoc across the server. For this reason, like PUT and PATCH, use of DELETE on a collection (or the main gateway of the resource: /users) should be disallowed or greatly limited.

When making a DELETE request, the client is instructing the server to permanently remove that item or collection.

When using a DELETE, you will most likely want to return one of three status codes. In the event that the item (or collection) has been deleted and you are returning a content or body response declaring such, you would

utilize status code 200. In the event that the item has been deleted and there is nothing to be returned back in the body, you would use status code 204. A third status code, 202, may be used if the server has accepted the request and queued the item for deletion but it has not yet been erased.

```
HTTP/1.1 204 No Content
cache-control: public, no-cache, no-store
Date: Tue, 31 Mar 2015 02:50:45 GMT
Connection: keep-alive
```

OPTIONS

Unlike the other HTTP Methods we've talked about, OPTIONS is not mappable to CRUD, as it is not designed to interact with the data. Instead, OPTIONS is designed to communicate to the client which of the methods or HTTP verbs are available to them on a given item or collection.

Because you may choose not to make every method available on each call they may make (for example not allowing DELETE on a collection, but allowing it on an item), the OPTIONS method provides an easy way for the client to query the server to obtain a quick list of the methods it is allowed to use for that collection or item.

```
HTTP/1.1 204 No Content
Allow: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE
Date: Tue, 31 Mar 2015 02:54:04 GMT
Connection: keep-alive
```

When responding to the OPTIONS method, you should return back either a 200 (if providing additional information in the body) or a 204 (if not providing any data outside of the header fields) unless, ironically, you choose not to implement the OPTIONS method, which would result in a 405 (Method Not Allowed) error. However, given that the purpose of the OPTIONS method is

to declare which methods *are* available for use, I would highly recommend implementing it.

Just the same, this is perhaps one of the most underused methods of the six. You can learn more about this—and the other HTTP methods—by visiting http://bit.ly/HTTPMethods.

9

Handling Responses

Since APIs are designed to be consumed, it is important to make sure that the client, or consumer, is able to quickly implement your API and understand what is happening. Unfortunately, many APIs make implementation extremely difficult, defeating their very purpose. As you build out your API you want to ensure that you not only provide informational documentation to help your developers integrate/ debug connections, but also return back relevant data whenever a user makes a call—especially a call that fails.

While having a well formatted, coherent body response is extremely important (you want something that can easily be deserialized, iterated and understood), you'll also want to provide developers with quick references as to what happened with the call, including the use of status codes. And in the case of a failure, you will want to provide descriptive error messages that tell the client not just what went wrong, but how to fix it.

HTTP Status Codes

When implementing REST over HTTP, we are able to take advantage of the HTTP Status Codes, a structure that most developers are familiar with, (For example, most developers can tell you that 200 is “okay,” 404 is “page/resource not found” and 500 is a server error.

Using the current HTTP status codes prevents us from having to create a new system that developers must learn, and creates a standard of responses across multiple APIs, letting developers easily integrate your API with others while using the same checks.

It is important, however, that you stick to standardized or accepted status codes, as you’ll find plenty of status codes in use across APIs that do not really exist. This creates the opportunity for confusion, for example, as Twitter uses its famous 420 status code (Enhance Your Calm) for too many requests, while Java’s Spring Framework used to return 420 to refer to a method failure (Now Deprecated).

In this use case Twitter, while opting for a humorous “Easter Egg” response, could have instead opted for status code 429 (Too Many Requests), and it may have been wiser for Spring Framework to return a 500 to represent a generic server error.

As you can see, someone utilizing the Spring Framework at this time while making a call to Twitter might be confused by what 420 really meant, and whether the method was not allowed (405) or there was a server error (500) instead of realizing they simply made too many calls. Imagine how much time and energy that confusion could cause them in debugging and trying to fix an application that is already working perfectly.

It’s also important to use status codes because the behavior of the server may be different from the expected behavior of the client. For example, if a client does a PUT on an item and the item doesn’t exist, per the RFC the item/object can then be created on the server—but not all APIs adhere to

this idea. As such, by returning a 201 (Created) instead of a 200 (OK), the client knows that the item did not previously exist and can choose to delete it (if it was accidentally created) or update their system with the data to keep everything in sync. Likewise, a 304 response would inform them that nothing was modified (maybe the data was identical), and a 400 would inform them that it was a bad request that could not be handled by the server (in the event where you elect not to create the item).

Some of the more common HTTP status codes you may run into, and should consider using, include the following:

Status Code	Definition
1XX	**Informational**
101	Switching Protocols (if moving the client from HTTP to another, less common, protocol)
2XX	**Successful**
200	Ok – The request has succeeded.
201	Created (e.g. a new resource or object within a collection)
202	Accepted – Your request has been accepted but has not yet been completed (e.g. delayed creates).
204	No Content – The request was successful, but there's nothing being returned in the body.
3XX	**Redirection**
301	Resource Moved Permanently
304	Not Modified (Nothing was modified by the request, e.g. PUT/PATCH)

307	Resource Moved Temporarily
4XX	**Client Error**
400	Bad Request – The request could not be understood or performed by the server due to malformed syntax and requires modifications before it can be performed.
401	Unauthorized – Authorization credentials are required, or the user does not have access to the resource/method they are attempting.
403	Forbidden – The request is understood but is being refused.
404	Resource not found – The URI is not recognized by the server.
405	Method not allowed – The attempted method is not allowed on the collection/item being called upon.
408	Timed out – The client did not provide information within the allotted timeframe.
409	Conflict – Unable to perform the action, usually on a PUT due to incompatible data sets (think of a GIT conflict that needs to be manually fixed before merging)
410	Gone – A resource or item that previously existed was permanently deleted and is no longer available for access. In the event it is unknown whether or not the deletion is permanent, or if the item pre-existed, a 404 should be used instead.
413	Request entity too long – The content the server is being asked to process is too large and must be broken down or resubmitted in a shorter/more compact format.
414	Request URI too long – The URI provided is too long and is not acceptable by the server. This is a fairly common error when trying to lengthen calls on Google's MAP API

415	Unsupported Media Type – The requested media type (content-type) is not supported. For example, if the user tries to submit a request using XML but you only support JSON.
429	Too many requests – The client has sent too many requests to the server in the given timeframe.

5XX	**Server Error**
500	The server ran into an unexpected error and was not able to perform the requested action (the generic "something went wrong on our end" error)
501	Not Implemented – Used when the client sends a request that the server is incapable of fulfilling due to lack of functionality
503	Service Unavailable – The server is temporarily unable to respond to the request (for example, due to maintenance) and the client should try again shortly.

Of course, there are more HTTP status codes you can take advantage of if desired, one being the popular 418 (I'm a Teapot). However, this 1998 April Fool's joke—while now an official HTTP Status Code—should only be used if the responding server is, in fact, a teapot. And while not quite as common as the other status codes, as more and more teapots become connected to the Internet, perhaps this is a code we'll be seeing a lot more of!

Handling Errors

Unfortunately, no matter how hard you try and how carefully you document your API, errors will always be a reality. Developers will skip over documentation, misunderstand it or simply discover that calls which were previously valid no longer work.

Because APIs are designed to be consumed, it is vital that you provide your clients with the tools, resources and information to help consume them. This means investing in your error messaging, an aspect of APIs that is often overlooked. In many cases companies have opted for generic error messages, as they have failed to consider the long-term ramifications.

Generic error messages tell developers that "something went wrong," but fails to tell them exactly what went wrong and how to fix it. This means that developers must spend hours debugging their code—and your API—in hopes of finding the answer. Eventually they'll either just give up (often finding a competitor's API instead) or contact support, requiring them to go through everything to track down oftentimes veiled and abstruse issues.

In the end, it costs you far more to have generic error messages than it does to provide descriptive error messages that alleviate developer frustration ("Oh, I know exactly what happened"); reduce development, debug, and integration time; and reduce support requirements. And by having descriptive error messages, when support is needed, they will have a good idea of what the issue is and where to look, saving you resources in that department as well (and keeping your API support team happy).

Bad Error Messages	Good Error Messages
Restricted Access	Invalid API Key/Access Token. If you need an API key, you can register for one at http://developers.mydomain.com/register
Permission Denied	Your API Key does not have access to this resource/ or the method attempted on this resource. Please contact our support team if you need access to this resource/ method.

Request Failed	The request was missing required data such as first name, last name email, etc. Additional detail may be found in our documentation by using the link provided below.

An additional advantage of providing descriptive error messages is that generic messages tend to be ambiguous and confusing. For example, what is the difference between “restricted access” and “permission denied?” Well, in the above cases, quite a bit. But someone unfamiliar with your API or naming conventions may not realize that.

A descriptive error should include an identifier for support (something that is short, such as a code that they can ask for and be able to look up quickly within their system), a description of what went wrong (as specific as possible), and a link to documentation where the developer can read more on the error and how to fix it.

Descriptive Error Formats

Thankfully, despite their fairly rare usage, descriptive error messaging is nothing new, and there are several different formats out there that already incorporate the above information, providing an easy way to implement descriptive errors in a standardized and recognized format. Three of the most popular ones include JSON API, Google Errors, and error.vnd.

JSON API

JSON API was created to serve as a way of returning back JSON-based response metadata, including hypertext links (which we’ll discuss in Chapter 12), as well as handling error bodies.

Rather than returning back just a single error, JSON API, as well as the other error specs we'll look at, lets you return back multiple errors in a single response, each containing a unique identifier, an error code to the correlating HTTP status, a brief title, a more in-depth message describing the error, and a link to retrieve more information.

```
{
   "error": {
      "errors": [{
         "id": "error1_firstName",
         "code": "XB500",
         "status": "400",
         "title": "User first name cannot be empty",
         "detail": "The first name field is required to
         have a value",
         "href": "http://docs.domain.ext/users/post"
      },
      {
         "id": "error2_lastName",
         "code": "XB501",
         "status": "400",
         "title": "User last name cannot be empty",
         "detail": "The last name field is required to
         have a value",
         "href": "http://docs.domain.ext/users/post"
      }]
   }
}
```

It is important to note that with JSON API errors, no other body data should be returned, meaning that the error should be the primary response and not embedded or returned with other resource body content or JSON.

JSON API Errors allow for the following properties to be used:

Property	Usage
error.errors	An array containing all of the errors that occurred (For example, if the form failed because of missing data, you could list out which fields are missing here with an error message for each of them.)
error.errors[].id	A unique identifier for the specific instance of the error (Please note that this is not the status code or an application-specific identifier.)
error.errors[].href	A URL that the developer can use to learn more about the issue (for example, a link to documentation on the issue)
error.errors[].status	The HTTP status code related to this specific error, if applicable
error.errors[].code	An application-specific code that identifies the error for logging or support purposes
error.errors[].title	A short, human-readable message that briefly describes the error
error.errors[].detail	A more in-depth, human-readable description of the error and how to resolve it

Additional information regarding JSON API errors may be found at http://jsonapi.org/format/#errors

Google Errors

Google Errors is the error handling format Google created for its own APIs, and is made up of two parts—a generalized error code and message, and an array of what caused the generalized error message—letting you break down exactly what went wrong and what the developer needs to fix. Another advantage to Google Errors is that it is property-, not format-based, meaning that it can be used across multiple formats including XML and JSON.

Within the errors array, the Google error format lets you return back the domain (area of the API) where the error occurred, the primary reason, a human-readable message, the location (based on setting the locationType property) and a URL for additional information.

For example, you can describe errors in JSON as shown below:

```
{
   "error": {
      "code": 400,
      "message": "The user was missing required fields",
      "errors": [{
         "domain": "global",
         "reason": "MissingParameter",
         "message": "User first name cannot be empty",
         "locationType": "parameter",
         "location": "firstName",
         "extendedHelp":
         "http://docs.domain.ext/users/post"
      },
      {
         "domain": "global",
         "reason": "MissingParameter",
         "message": "User last name cannot be empty",
         "locationType": "parameter",
```

```
            "location": "lastName",
            "extendedHelp":
            "http://docs.domain.ext/users/post"
         }]
      }
   }
```

Or as described in an XML format:

```
<?xml version="1.0" encoding="UTF-8" ?>
<errors>
   <code>400</code>
   <message>The user was missing required fields</message>
   <error>
      <domain>global</domain>
      <reason>MissingParameter</reason>
      <message>User first name cannot be empty</message>
      <location type="parameter">firstName</location>
      <extendedHelp>http://docs.domain.ext/users/post</ext
      endedHelp>
   </error>
   <error>
      <domain>global</domain>
      <reason>MissingParameter</reason>
      <message>User last name cannot be empty</message>
      <location type="parameter">lastName</location>
      <extendedHelp>http://docs.domain.ext/users/post</ext
      endedHelp>
   </error>
</errors>
```

Google Errors allows for the following properties to be used:

Property	Usage
error.code	An integer representing the HTTP status error code
error.message	A brief description of what the overall, generalized error is. You will be able to explain exactly what caused the overall error in the errors array later.
error.errors	An array containing all of the errors that occurred. For example, if the form failed because of missing data, you could list out which fields are missing here with an error message for each of them.
error.errors[].domain	The name of the service, class or other identifier for where the error occurred to better provide an understanding of what the issue was
error.errors[].reason	A unique identifier for the error, or the specific type of error. For example, “InvalidParameter.”
error.errors[].message	A description of the error in a human-readable format.
error.errors[].location	The location of the error with interpretation determined by the location type (In the case of a parameter error, it might be the form field where the error occurred, such as “firstName.”)
error.errors[].locationType	Determines how the client should interpret the location property (For example, if the location is specific to a form field/ query parameter, you would use “parameter” to describe the locationType.)
error.errors[].extendedHelp	A URL that the developer can use to learn more about the issue (For example, a link to documentation on the issue.)
error.errors[].sendReport	A URI to be used to report the issue if the developer believes it was caused in error and/or to share data regarding the error.

Additional information regarding Google Errors may be found at http://bit.ly/1wUGinJ

vnd.error

Created by Ben Longden in 2012, vnd.error is a concise JSON error description format that allows for embedding and nesting errors within a response. It was specifically designed to be fully compatible with HAL-based hypertext linking format (we'll discuss HAL in the next chapter) and is expressed by using the content-type: application/vnd.error+json.

Unlike JSON API errors, a vnd.error can be nested within a larger response. However, in the case of an error, this is most likely not recommended, as you would not want the client to continue the request before fixing any client-based issues.

vnd,error is also far more concise in its response, returning back a message, an error code identifier and a URL the developer may use to learn more about the issue.

```
{
    "total": 2,
    "_embedded": {
        "errors": [ {
            "message": “User first name cannot be empty",
            "logref": “XB500”,
            "_links": {
                "help": { "href": "http://docs.domain.ext/users/post" }
            }
        },
        {
            "message": “User last name cannot be empty",
            "logref": “XB501”,
            "_links": {
                "help": { "href": "http://docs.domain.ext/users/post" }
            }
        }]
    }
}
```

vnd.error allows for the following properties to be used:

Property	Usage
errors[]	An array containing all of the errors that occurred (For example, if the form failed because of missing data, you could list out which fields are missing with an error message for each of them.)
errors[].message	A description of the error in a human-readable format.
errors[].logref	A unique identifier for the error or the specific type of error (For example “InvalidParameter” to be used for logging purposes)
errors[].path	The resource for which the error is relevant. This becomes useful when using the “about” property in the links section when an error can apply to multiple resources.
errors[].links[].help	A URL that the developer can use to learn more about the issue (For example, a link to documentation on the issue)
errors[].links[].describes	The correlating server side error that the response error is describing, if applicable (See RFC6892.)
errors[].links[].about	A link to the resource to which the error applies (See RFC6903.)

Additional information regarding vnd.error may be found at https://github.com/blongden/vnd.error

Usability is Key

Remember, one of the keys to your API's success is usability. It's easy to return a "Permission Denied" error to a resource, but it is far more helpful if you return "Invalid Access Token" with additional resources telling the developer exactly what the problem is.

Making your API easy to consume with HTTP Status Codes and Descriptive Error Messaging may be the difference between an award-winning application on your API or an award-winning application on your competitor's.

10

Adding Hypermedia

In his dissertation, Dr. Fielding described four sub-constraints of the Uniform Interface, one being Hypermedia as the Engine of Application State. Unfortunately, while a common occurrence in the computer industry, hypermedia has presented a challenge in both building a hypermedia-driven API and consuming it.

Before we can truly work to incorporate hypermedia into our API, we must first understand what it is. The term, coined back in 1965 by Ted Nelson, is simply an expansion on hypertext—something you may recognize as part of today's World Wide Web.

In fact, we use hypertext every single day, as it is a key component in the Hypertext Markup Language (HTML). This means that chances are—possibly without even realizing it—you have been not only using, but also writing hypertext and hypermedia on a frequent basis.

To put it succinctly, hypertext is simply regular text with links to other documents. By adding these links, we create a structured document that users can access and browse, guiding them through the possible actions and relevant resources.

For example, if you visit the Yahoo! home page, you'll see several different links for different sections, including the news. If you click on "News," it takes you to a whole new page, which loads up hundreds of more links directing you to different news articles. In the most basic sense, this is hypertext linking at work. The links guide you to where you want to go and what your possible actions from within that area are.

In the same way, the easiest way to incorporate these links (or hypertext/hypermedia) is to incorporate links within your API response. Or, as Keith Casey explains it, "provide your users with a 'choose their own adventure' book." It's a way for them to say, "I want to take this action; now what are the available actions based on that?"

Take for example, if in our API we had created a new user, we might get back a response that looks something like this:

```
{
   "users": [
      {
         "id": 1,
         "firstName": "Mike",
         "lastName": "Stowe",

         "_links": {
             "update": {
                 "description": "edit the user",
                 "href": "/api/users/1",
                 "methods": ["put","patch"]
             },
```

```
            "delete": {
                "description": "delete the user",
                "href": "/api/users/1",
                "methods": ["delete"]
            },
            "message": {
                "description": "message the user",
                "href": "/api/message/users/1",
                "methods": ["post"]
            }
        }
    }
  ]
}
```

You'll notice that within the user response we have a section called _links, in this case taking advantage of CPHL, or the Cross Platform Hypertext format (experimental). Each one of these links tells the client which resources are available to them based on the user they just created. For example, they can now edit the user, delete the user, message the user, view the user's timeline and post to the user's timeline. Different hypertext specs may look and handle this slightly differently, but ultimately they all tell us which resources are available through links.

Along with directing us to the possible actions of an item, hypertext links can also make navigating collections easier, especially in regards to pagination.

```
{
    "_links": {
        "beginning": {
            "description": "first page",
            "href": "/api/users?page=1",
            "methods": ["get"]
```

```
        },
        "prev": {
            "description": "previous page",
            "href": "/api/users?page=4",
            "methods": ["get"]
        },
        "next": {
            "description": "next page",
            "href": "/api/users?page=6",
            "methods": ["get"]
        },
        "last": {
            "description": "last page",
            "href": "/api/users?page=9",
            "methods": ["get"]
        }
    }
}
```

Now, adding hypermedia/hypertext links to an API can be scary for several reasons. First, all one has to do is search for something called HATEOAS, or Hypermedia as the Engine of Application of State, to find complex explanations and diagrams. On the flip side, you'll also find arguments about the lack of a "hypermedia client," among others.

I think its important to point out right off the bat that hypermedia is neither a cure-all nor a perfect solution. Part of the reason is that while it's been around since 1965, it's still relatively new for RESTful APIs. As such, we're still working on trying to figure out the exact formula, and new ideas are being tested every day.

However, that doesn't mean that you shouldn't understand how hypermedia works in a RESTful API, or implement it in your own. While there are plenty of challenges (which we'll take a look at shortly), there are

far more benefits, and by using hypermedia within your API, you will increase its longevity and long-term usability.

Hypermedia as the Engine of Application State

Perhaps the most daunting of terms is Hypermedia as the Engine of Application State (HATEOAS), over which the debate is endless. In fact, today there are still arguments over how to pronounce it. Is it “HATE-E-OSS”? “Hat-E-OSS”? “Hat-E-AS”? The list goes on.

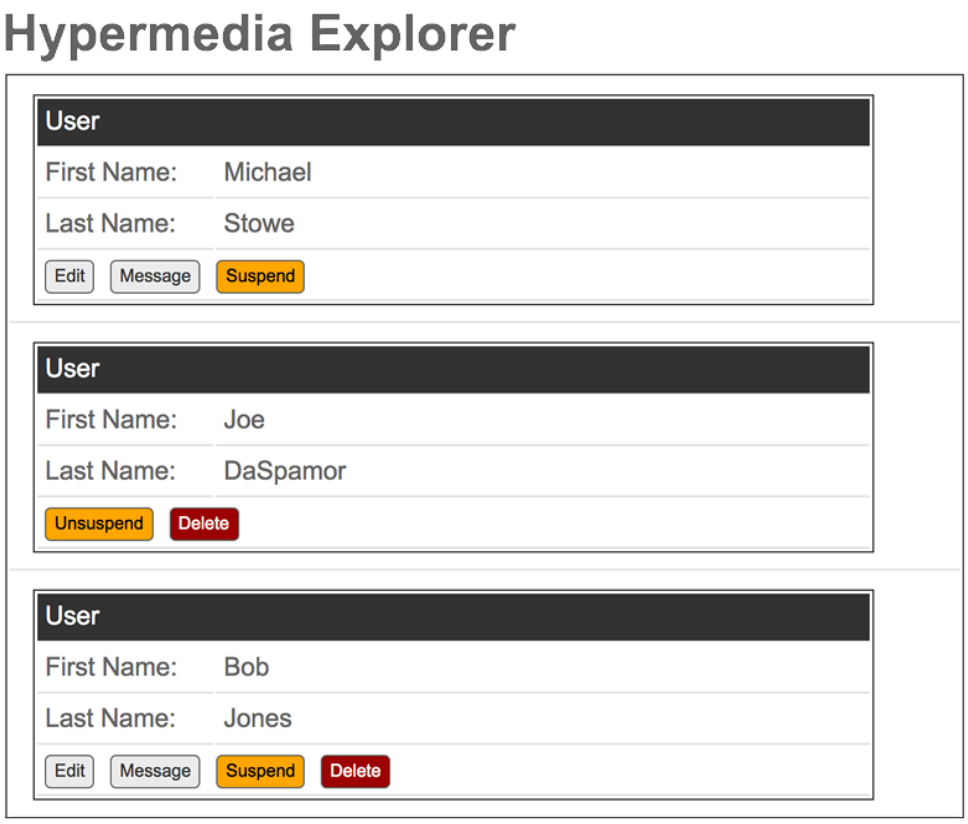
To make the concept more difficult, Fielding doesn’t explain what HATEOAS is in his dissertation, simply saying that it’s a key component to having a uniform interface. While this has started some debates (as there always are in computational theories), the easiest way to describe HATEOAS is this—for every object you have, you have possible actions that can be applied to it. However, these actions may not be applicable to all objects in a collection, as all objects are not necessarily the same.

In other words, if you have a collection of users, you may have users who haven’t confirmed their email addresses yet, standard users, admins, superusers, etc. Because each user is unique, the actions you can take on each user might also be unique. For example, you might have a link to confirm an unconfirmed email user’s email address, but not have that action for other users. You may have admin links that you will provide to your admin and superuser, but not to your standard users.

The real question is, how do you know which actions are available for which user when those actions are dynamic? How do you know which actions a user has access to if he or she is a superuser?

Now, you can have the client try to hardcode these rules into their system, check to determine the user’s role, and then assume those actions are available, or you can communicate this to the client through the API in real-

time. As complex as it sounds, this is what it means to use hypermedia as the engine of application state.



Wait—I know what you're thinking! REST is supposed to be stateless! And it is; with REST, all state is handled by the client. That means the server does not keep track of the calls the client is making, but rather receives one call and returns the data based on the call it received. With HATEOAS, you are not changing this. Instead, you are providing them with a representation of the object's or application's state (in other words, which actions they can take with the item or collection they have).

This prevents the client from having to guess and hardcode your application's rules into their own application, and instead lets them rely on *your* rules and the data in real-time. Rather than having clients making calls for actions they cannot do, by implementing hypermedia, you are telling them the state of that item and what it is they *can* do.

As the state of that item changes, so does the response they receive back from the server—again, represented by the hypertext links that are included within the response.

Implementing Hypermedia

Of course, one of the main arguments against hypermedia is it requires taking substantial amounts of time to think of every possible action a person could take within an item or collection—something that is absolutely true. In order to provide the most helpful and complete set of links, you will need to know how your API interacts with itself—something you should already have a good grasp on.

And if by chance you started building out your API as described in Chapter 2, you already have the linking relationships defined:

Users	Create a user, edit a user, retrieve username, retrieve password, reset password, view profile, **Message user**
Messages	**Send a message**, create a draft, send a draft, delete draft, get message, mark message as read, mark message as unread, move message to folder, delete message
Products	View product, review product, **add product to cart,** add to wishlist
Cart	View cart, **add product**, change quantity, delete product, checkout

As simple as this chart may seem, it helps us identify the different resources that might need to interact with each other and helps us eliminate the need to go back and locate the different resources and/or create complex diagrams or charts to define them. Instead, we can simply look at the "Users" section, which we will assume became our "/users" resource, and we now know which actions to which the client may need access, or which links we should provide (based on our application's rules, of course).

Using the above chart, we can quickly determine that there are multiple actions within the /users resource to which they may need access (and it's quite possible some of these are their own resources), but they would also need access to the URI for sending a message, which we determined in Chapter 2 belonged in the /messages resource.

Essentially, we've done the hard work already. Now we just need to add these links into our response using a flexible architecture and, preferably, one of the specs below.

Common Hypermedia Specs

Thankfully, to make Hypermedia implementation even easier, there are linking specifications already in existence. Each specification is slightly different, with some offering broader language support, some focusing on actions over resources, and some incorporating documentation and form data.

Unfortunately, none of these are perfect, and each specification has its own strengths and weaknesses. While HAL and JSON API are the clear choice for most hypermedia implementations today, others (such as PayPal) have elected to create their own specification to meet their needs. For this reason, it's important to look at how you want developers interacting with your API and which spec best meets their—and your—needs.

Collection+JSON http://bit.ly/17P84eR

Collection+JSON is a JSON-based read/write hypermedia-type designed by Mike Amundsen back in 2011 to support the management and querying of simple collections. It's based on the Atom Publication and Syndication specs, defining both in a single spec and supporting simple queries through the use of templates. While originally widely used among APIs, Collection+JSON has struggled to maintain its popularity against JSON API and HAL.

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/friends/",

    "links" : [
      {"rel"        :        "feed",        "href"        :
"http://example.org/friends/rss"},
      {"rel"       :       "queries",       "href"       :
"http://example.org/friends/?queries"},
      {"rel"       :       "template",       "href"       :
"http://example.org/friends/?template"}
    ],

    "items" : [
      {
        "href" : "http://example.org/friends/jdoe",
        "data" : [
          {"name"  :  "full-name",  "value"  :  "J.  Doe",
"prompt" : "Full Name"},
          {"name" : "email", "value" : "jdoe@example.org",
"prompt" : "Email"}
        ],
        "links" : [
```

```
          {"rel"       :       "blog",       "href"       :
"http://examples.org/blogs/jdoe", "prompt" : "Blog"},
          {"rel"       :       "avatar",       "href"       :
"http://examples.org/images/jdoe",  "prompt"  :  "Avatar",
"render" : "image"}
        ]
      }
    ]
  }
}
```

Strengths:	**Strong choice for collections, templated queries, early wide adoption, recognized as a standard**
Weaknesses:	JSON only, lack of identifier for documentation, more complex/ difficult to implement

JSON API http://jsonapi.org/

JSON API is a newer spec created in 2013 by Steve Klabnik and Yahuda Klaz. It was designed to ensure separation between clients and servers (an important aspect of REST) while also minimizing the number of requests without compromising readability, flexibility or discovery. JSON API has quickly become a favorite, receiving wide adoption, and is arguably one of the leading specs for JSON-based RESTful APIs. JSON API currently bears a warning that it is a work in progress, and while widely adopted, it is not necessarily stable.

```
{
  "links": {
    "posts.author": {
      "href": "http://example.com/people/{posts.author}",
```

```
      "type": "people"
    },
    "posts.comments": {
      "href":
"http://example.com/comments/{posts.comments}",
      "type": "comments"
    }
  },
  "posts": [{
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "9",
      "comments": [ "5", "12", "17", "20" ]
    }
  }]
}
```

Strengths:	**Simple versatile format, easy to read/implement, flat link grouping, URL templating, wide adoption, strong community, recognized as a hypermedia standard**
Weaknesses:	JSON-only, lack of identifier for documentation, still a work in progress

HAL http://bit.ly/1ELp7LP

HAL is an older spec, created in 2011 by Mike Kelly to be easily consumed across multiple formats including XML and JSON. One of the key strengths of HAL is that it is nestable, meaning that _links can be incorporated within each item of a collection. HAL also incorporates CURIES, a feature that makes it unique in that it allows for inclusion of documentation links in the

response, though they are tightly coupled to the link name. HAL is one of the most supported and widely used hypermedia specs out there today, and is supported by a strong and vocal community.

```
{
    "_links": {
        "self": { "href": "/orders" },
        "curies":     [{     "name":     "ea",     "href":
"http://example.com/docs/rels/{rel}",   "templated":   true
}],
        "next": { "href": "/orders?page=2" },
        "ea:find": {
            "href": "/orders{?id}",
            "templated": true
        },
        "ea:admin": [{
            "href": "/admins/2",
            "title": "Fred"
        }, {
            "href": "/admins/5",
            "title": "Kate"
        }]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20,
    "_embedded": {
        "ea:order": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "ea:basket": { "href": "/baskets/98712" },
                "ea:customer": { "href": "/customers/7809"
}
            },
            "total": 30.00,
```

```
            "currency": "USD",
            "status": "shipped"
        }, {
            "_links": {
                "self": { "href": "/orders/124" },
                "ea:basket": { "href": "/baskets/97213" },
                "ea:customer":          {          "href":
"/customers/12369" }
            },
            "total": 20.00,
            "currency": "USD",
            "status": "processing"
        }]
    }
}
```

Strengths:	**Dynamic, nestable, easy to read/implement, multi-format, URL templating, inclusion of documentation, wide adoption, strong community, recognized as a standard hypermedia spec, RFC proposed**
Weaknesses:	JSON/XML formats architecturally different, CURIEs are tightly coupled

JSON-LD http://json-ld.org/

JSON-LD is a lightweight spec focused on machine-to-machine readable data. Beyond just RESTful APIs, JSON-LD was also designed to be utilized within non-structured or NoSQL databases such as MongoDB or CouchDB. Developed by the W3C JSON-LD Community group, and formally recommended by W3C as a JSON data linking spec in early 2014, the spec has struggled to keep pace with JSON API and HAL. However, it has built a

strong community around it with a fairly active mailing list, weekly meetings and an active IRC channel.

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Strengths:	**Strong format for data linking, can be used across multiple data formats (Web API and databases), strong community, large working group, recognized by W3C as a standard**
Weaknesses:	JSON-only, more complex to integrate/interpret, no identifier for documentation

CPHL http://bit.ly/18iXlcC

The spec I created in 2014, CPHL, or the Cross Platform Hypertext Language, is an experimental specification based on HAL that incorporates methods, documentation, code on demand, and API definition specifications (such as RAML or Swagger). Unlike the other specifications, CPHL is designed to be action-first, not resource-first, by focusing on what the link does verses what resource it is. CPHL is also designed to provide an architecturally similar structure across different formats such as XML, YAML and JSON, making it easier to incorporate in an application after deserialization. Just the same, CPHL is listed as a brainstorming document, is not stable, and does not have a strong community or broad adoption.

```
{
    "_definition": {
        "raml": "http://api.domain.com/docs/api/raml",
        "swagger":
"http://api.domain.com/docs/api/swagger"
    },
    "_links": {
        "update": {
            "title": "Edit User",
            "description": "edit the user",
            "href": "/api/resource",
            "methods": [
                "put",
                "patch"
            ],
            "formats": {
                "json": {
                    "mimeType": "application/json",
                    "schema":
"http://api.domain.com/docs/api/editSchema.json"
                },
                "xml": {
                    "mimeType": "text/xml",
                    "schema":
"http://api.domain.com/docs/api/editSchema.xml"
                }
            },
            "docHref": "http://api.domain.com/docs/edit",
            "code": {
                "php": {
                    "href":
"http://code.domain.com/phplib/edit.tgz",
                    "md5":
"0cc175b9c0f1b6a831c399e269772661",
```

```
                    "recordSpecific": false
                },
                "java": {
                    "href":
"http://code.domain.com/javalib/edit.tgz",
                    "md5":
"0cc175b9c0f1b6a831c399e269772661",
                    "recordSpecific": false
                },
                "ruby": {
                    "href":
"http://code.domain.com/rubylib/edit.tgz",
                    "md5":
"0cc175b9c0f1b6a831c399e269772661",
                    "recordSpecific": false
                }
            }
        }
    }
}
```

Strengths:	**Designed for cross-platform consistency; allows loosely-coupled documentation; incorporates API definitions, methods and code on demand; and allows for multiple formats while also providing a strict naming structure for common actions**
Weaknesses:	Poor adoption/not heavily tested, can become bloated, work in progress, listed as a brainstorming document

Siren http://bit.ly/1F72aAK

Created in 2012 by Kevin Swiber, Siren is a more descriptive spec made up of classes, entities, actions and links. It was designed specifically for Web API clients in order to communicate entity information, actions for executing state transitions and client navigation/discoverability within the API. Siren was also designed to allow for sub-entities or nesting, as well as multiple formats including XML—although no example or documentation regarding XML usage has been provided. Despite being well intentioned and versatile, Siren has struggled to gain the same level of attention as JSON API and HAL. Siren is still listed as a work in progress.

```
{
  "class": [ "order" ],
  "properties": {
      "orderNumber": 42,
      "itemCount": 3,
      "status": "pending"
  },
  "entities": [
    {
      "class": [ "items", "collection" ],
      "rel": [ "http://x.io/rels/order-items" ],
      "href": "http://api.x.io/orders/42/items"
    },
    {
      "class": [ "info", "customer" ],
      "rel": [ "http://x.io/rels/customer" ],
      "properties": {
        "customerId": "pj123",
        "name": "Peter Joseph"
      },
      "links": [
        {     "rel":     [     "self"     ],     "href":
"http://api.x.io/customers/pj123" }
```

```
      ]
    }
  ],
  "actions": [
    {
      "name": "add-item",
      "title": "Add Item",
      "method": "POST",
      "href": "http://api.x.io/orders/42/items",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        {   "name":   "orderNumber",   "type":   "hidden",
"value": "42" },
        { "name": "productCode", "type": "text" },
        { "name": "quantity", "type": "number" }
      ]
    }
  ],
  "links": [
    {      "rel":      [      "self"      ],      "href":
"http://api.x.io/orders/42" },
    {     "rel":     [     "previous"     ],     "href":
"http://api.x.io/orders/41" },
    {      "rel":      [      "next"      ],      "href":
"http://api.x.io/orders/43" }
  ]
}
```

Strengths:	**Provides a more verbose spec, query templating and form fields; incorporates actions, multi-format**
Weaknesses:	Poor adoption, lacks documentation, work in progress

Other Specs

Because there is no standardized way of doing hypermedia, and because we still face challenges today in implementing hypermedia into our APIs and getting it to work exactly how we would like, new specs are being created every day. A quick look at the specs we just covered will show that of the six, half were created in the last two years.

Other newborn specs include Mike Amundsen's UBER, Jorn Wildt's Mason and Niels Krijger's Yahapi [Yet Another Hypermedia(ish) API Specification]. Each tries to challenge the way we look at hypermedia APIs and each, like CPHL, tries to find a better way of doing things.

However, while it's important to innovate when building your API, it's also important to adhere to the tools and standards that developers know, and the specifications that have been carefully tried and tested. While I would personally urge you to try specs such as CPHL, Siren, UBER, Yahapi and others in your own personal projects, I have to recommend utilizing HAL or JSON API for your production API unless you have a strong reason to choose a different specification.

Unfortunately, no hypermedia specification is perfect, as we will talk about in the next section.

Hypermedia Challenges

While hypermedia dominates the Internet through formats such as HTML, there are distinct advantages that are lacking in the API space, making the implementation and usage of hypermedia more challenging.

For example, unlike HTML, we have already listed several different mainstream specs that you can use to describe your links—with more being created every single day. That means that as of right now, there is no universal standard for how to format your links. In fact, many companies are even creating their own specs, forcing developers to learn and apply different formats to every API they utilize.

The other challenge is intelligence, as HTML—when interpreted by the browser—presents the links to humans for them to determine the next action they want to take. This allows for rather creative linking formats, as links can be placed wherever, and with whatever text the designer wants. API hypertext links, however, must be interpreted by machines, making it far more difficult to add creative linking throughout the response.

These two challenges alone greatly hinder the advancement and usage of hypermedia in APIs today. But again, even with these limitations, when used as the engine of application state, hypermedia presents a level of state that would not otherwise be available while also providing a layer of flexibility to your API. This benefit should not be discarded or overlooked because of these other challenges.

Client Resource URI Dependence

Another challenge with purely hypermedia-driven APIs as described by Fielding is that there is a single point of entry to determine which resources are available. While in certain cases this works extremely well (for example, with StormPath's API), but for more complex APIs with multiple pathways, this requires the client to make multiple calls to the API just to get to a starting point.

Many developers choose to hardcode URIs, either because they don't fully understand the benefits of a Hypermedia API, or they don't know that the API is not truly fluid (or that the resource they are calling shouldn't change). This is also one of the key arguments against the use of HATEOAS—that it may break resource URIs.

The benefit to this, however, is that it forces our API to act just like a browser, and when a URI is changed, we simply get the updated link and click on it. For example, going back to Yahoo!—when you visit, there are key URIs that you utilize, such as the base domain to access the site, and then all available links are presented to you. This means that you are able to get the latest information without having to remember the links, and that if a link changes, it does not negatively affect you unless you saved the link in your browser (i.e. hardcoded it).

This means that as your API continues to evolve, and as resources are added, changed, and removed, that in a purely hypermedia-driven API, your developers and clients would not be adversely affected (assuming the client is smart enough to handle the different paths/actions/responses).

Unfortunately, because hypermedia in the sense of APIs is so new, pioneering into the world of true hypermedia APIs presents substantial challenges and could negatively impact usage. As such, it is important to understand that regardless of whether or not you build a truly hypermedia driven API, unless you force your users to start at the base URI, they will most likely hardcode the resources that they want to use in their application. This means that regardless of the format in which you build your API, you should be wary about changing resource paths and understand that doing so—even in a hypermedia driven API—may have adverse and unintended effects.

The good news is that, again, this doesn't detract from the state benefit that HATEOAS brings with it, and as technology and usage of hypermedia

driven APIs increases, so will developers' understanding of its benefits and implementation. Just like REST was confusing for many developers at first, we should expect to see a learning curve as more and more developers consume hypermedia-driven APIs. This means that should you choose to build a hypermedia-driven API, as REST dictates, you should be prepared to educate your users through documentation, tutorials and examples.

One of the best ways to do this is through the use of the API Notebook, which lets you walk your users through exactly how the API works in any given scenario—something we will cover in Chapter 12.

11

Managing Your API with a Proxy

Once you have designed and built your API, one of the most crucial aspects is protecting your system's architecture and your users' data, and scaling your API in order to provide downtime and meet your clients demands.

The easiest and safest way to do this is by implementing an API manager, such as MuleSoft's API Gateway. The API manager can then handle API access (authentication/provisioning), throttling and rate limiting, setting up and handling SLA tiers and—of course—security.

A hosted API manager also provides an additional layer of separation, as it can stop DDoS attacks, malicious calls and over-the-limit users from ever reaching your system's architecture, while also scaling with demand—meaning the bad requests are killed at a superficial layer, while valid requests are passed through to your server.

Of course, you can build your own API manager and host it on a cloud service such as AWS, but you have to take into consideration both the magnitude and the importance of what you're building. Because an API manager is designed to provide both scalability and security, you'll need to make sure you have system architects who excel in both, as one mistake can cost hundreds of thousands—if not millions—of dollars.

And like any large-scale system architecture, trying to design your own API manager will most likely prove costly—usually several times the cost of using a third-party API manager when all is said and done.

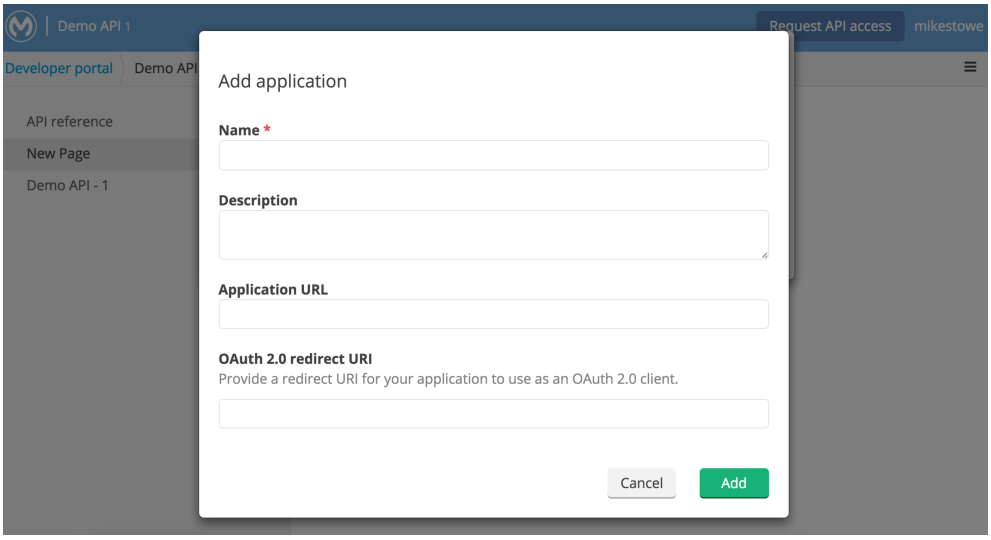
For this reason, I highly recommend choosing an established API management company such as MuleSoft to protect and scale your API, as well as provide you with the expertise to help your API continue to grow and live a long, healthy life.

API Access

Controlling access is crucial to the scale and security of your API. By requiring users to create accounts and obtain API keys or keys for their application, you retain control. The API key acts as an identifier, letting you know who is accessing your API, for what, how many times, and even what they are doing with that access.

By having an API key, you can monitor these behaviors to isolate malicious users or potential partners who may need a special API tier. If you choose to monetize your API, monitoring API access will also allow you to identify clients who may want or need to upgrade to a higher level.

Typically, an API key is generated when a user signs up through your developer portal and then adds an application that they will be integrating with your API.



However, this isn't always the case, as some companies create singular API keys that can be used across multiple applications. However, by setting it at the application level, you can see exactly the types of applications for which the client is utilizing your API, as well as which applications are making the most calls.

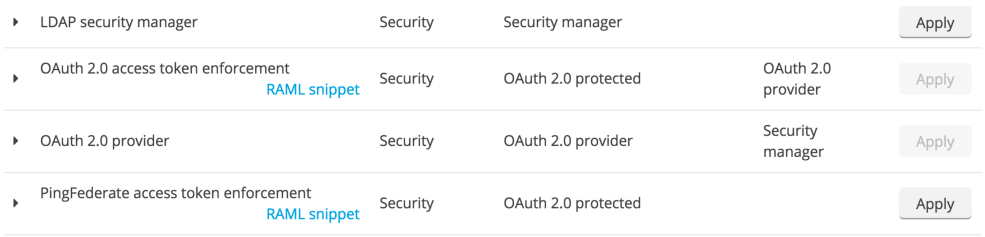
Application	Current SLA tier	Requested SLA tier	Status	
Mike's Application	Basic	N/A	Approved	Revoke

Owner	Mike Stowe michael.stowe@mulesoft.com	**Submitted**	7 months ago
Client ID	2b990d4783b9440d8dbfa78431825066	**Approved**	4 months ago
URL	http://www.mikestowe.com	**Rejected**	-
Redirect URL	http://www.mikestowe.com/oauth.php	**Revoked**	5 months ago

My App	partner	N/A	Approved	Revoke

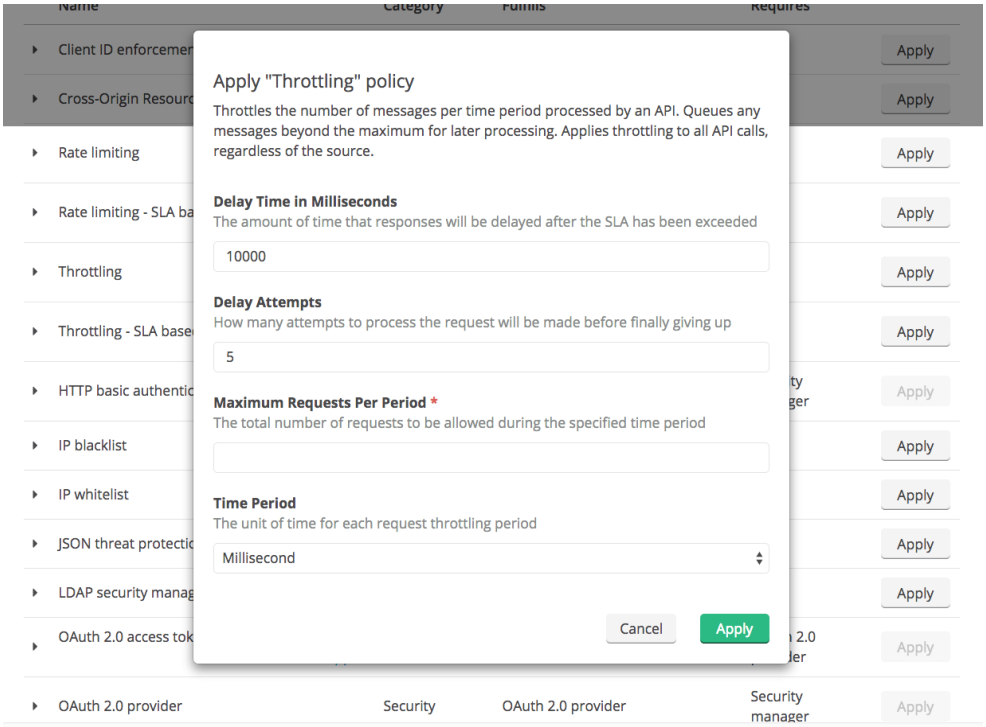
OAuth2 and More

A good API manager goes beyond just the basic management of API keys, and will also help you implement security for restricting access to user information, such as OAuth2, LDAP, and/ or PingFederate. This lets you take advantage of systems you are already utilizing, or the flexibility of using a third party service to handle OAuth if you choose not to build it yourself (remember chapter 6):

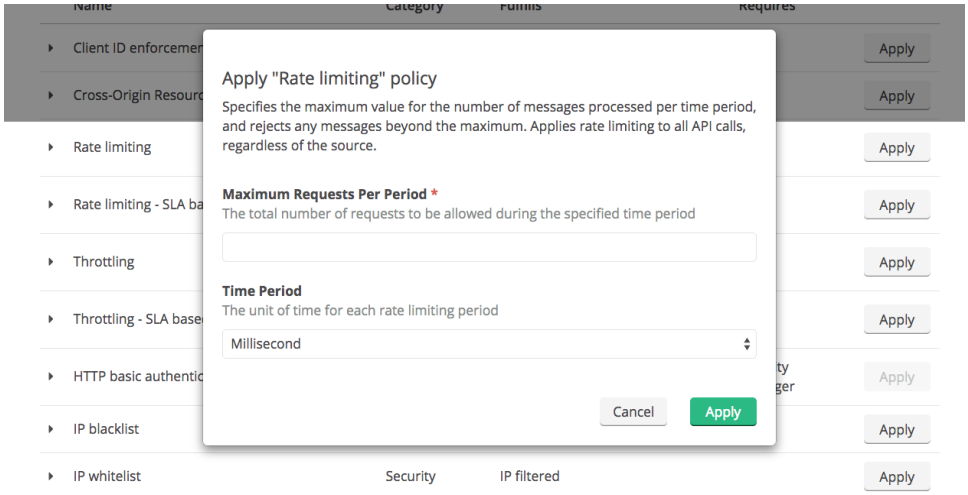


Throttling

Throttling and rate limiting allow you to prevent abuse of your API, and ensure that your API withstands large numbers of calls (including DoS and accidental loops). With Throttling, you can set a delay on calls after an SLA tier has been exceeded, slowing down the number of calls an API client is making.



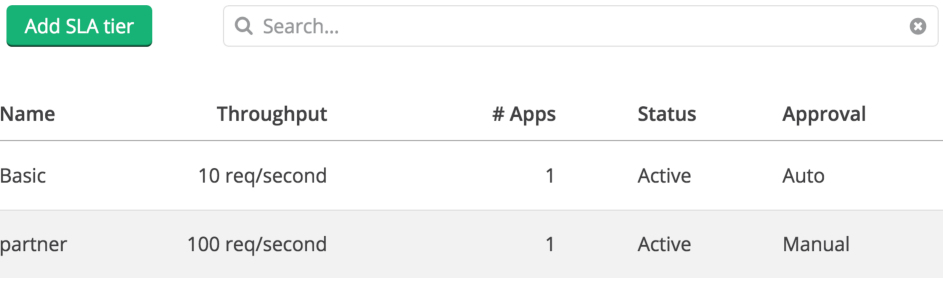
Rate limiting lets you set a hard number for how many calls the client may make in a specific time frame. Essentially, if a client is making too many calls, you can slow down the responses or cut the client off to prevent the system from being overrun or disrupting your other users.



This is especially helpful in negating malicious attacks, as well as the dreaded accidental infinite loop that pounds your API with calls. While this practice may seem harsh at first, it is widely adopted to ensure the best quality of service for everyone.

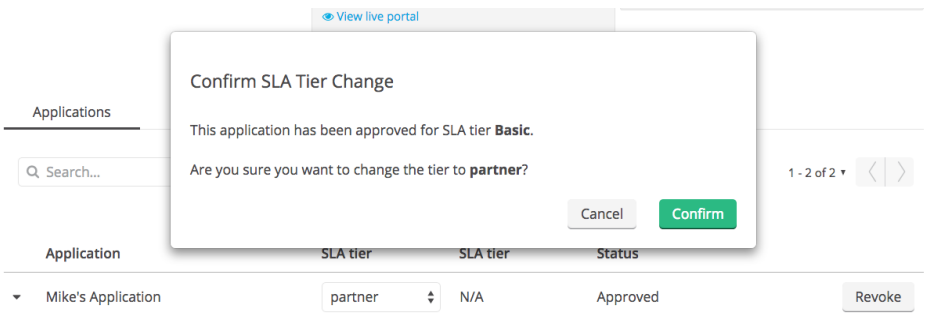
SLA Tiers

SLA tiers, or Service Level Agreements let you set up different rules for different groups of users. For example, you may have your own mobile apps, premium partners, paid API users, and standard/free users. You may want to limit the access of each of these groups to ensure the highest quality engagement for your users, while also helping prevent loops by inexperienced developers testing out your API. For example, you can ensure premium partners and your mobile apps have priority access to the API with the ability to make thousands of calls per second, while the standard API user may only need four calls per second. This ensures that the applications needing the most access can quickly obtain it without any downtime, while your standard users can also rely on your API without having to worry about someone abusing the system, whether accidentally or on purpose.



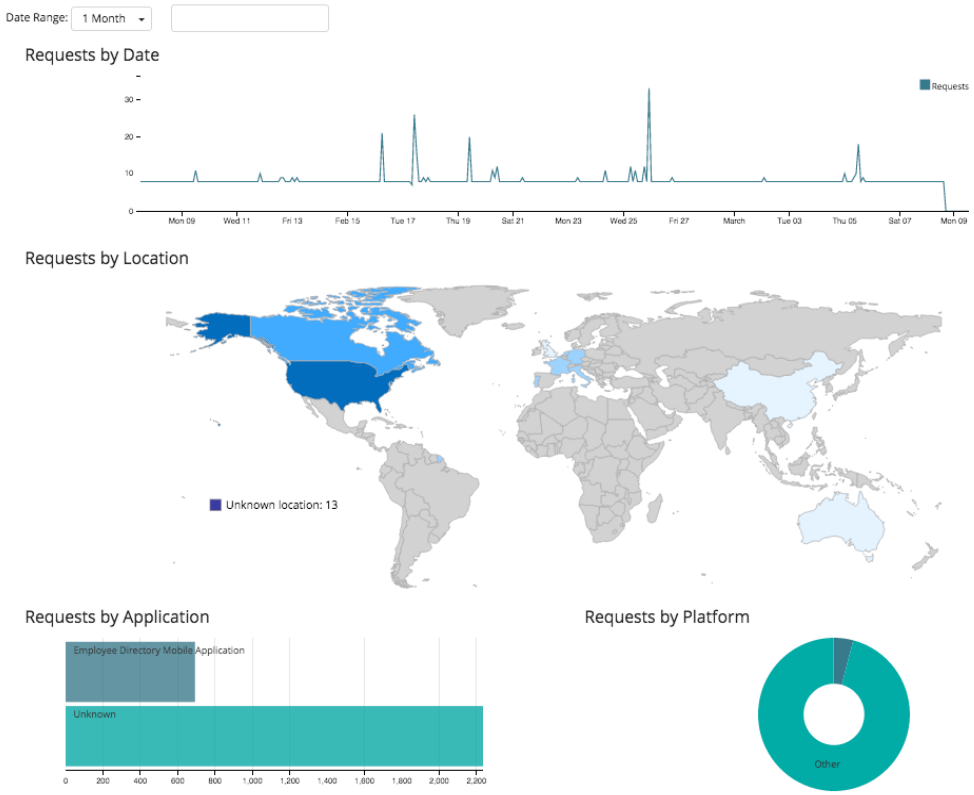
An API manager that incorporates SLA tiers should let you create the different levels and specify the number of requests per second users in this tier are allotted. You should also be able to determine whether or not users of the tier require your approval. For example, you may offer automatic approval to premium partners, but require special agreements and manual intervention in order for basic or free users to take advantage of the higher throughput.

Once you have set up your SLA tiers, you should be able to assign applications to that tier.



Analytics

Another valuable tool your API manager should provide you is analytics, letting you quickly see which of your APIs are the most popular and where your calls are coming from. These analytics can help you identify which types of devices (OS) the clients are using, the top applications using your API and the geographical location of those accessing your API.



Along with identifying your API's usage trends, as well as being able to monitor API uptime/spikes/response times (especially in regards to your own server architecture), analytics also help you prove the business use

case for your API, as you are able to show both its popularity and how it is being used.

These metrics are especially key when making business decisions, reporting to business owners/stakeholders, and designing a Developer Relations/Evangelism program (as often times API Evangelist's performances are based on the number of API keys created and the number of calls to an API).

Security

Security is an essential element of any application, especially in regards to APIs, where you have hundreds, to thousands, to hundreds of thousands of applications making calls on a daily basis.

Every day, new threats and vulnerabilities are created, and every day, companies find themselves racing against the clock to patch them. Thankfully, while an API manager doesn't eliminate all threats, it can help protect you against some of the most common ones. And when used as a proxy, it can prevent malicious attacks from hitting your architecture.

It's important to understand that when it comes to security, you can pay a little up front now, or a lot later. After all, according to Stormpath, in 2013 the average cost of a personal information breach was $5.5 *million*. When Sony's PlayStation network was hacked, exposing 77 million records, the estimated cost to Sony was $171 million for insurance, customer support, rebuilding user management and security systems.

This is in part why you should seriously consider using a pre-established, tested API manager instead of trying to build your own, because not only do you have the development costs, but also the risks that go along with it. When it comes to security, if you don't have the expertise in building these types of systems, it's always best to let those with expertise do it for you.

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing, or CORS, allows resources (such as JavaScript) to be used by a client outside of the host domain. By default, most browsers have strict policies to prevent cross-domain HTTP requests via JavaScript due to the security risks they can pose.

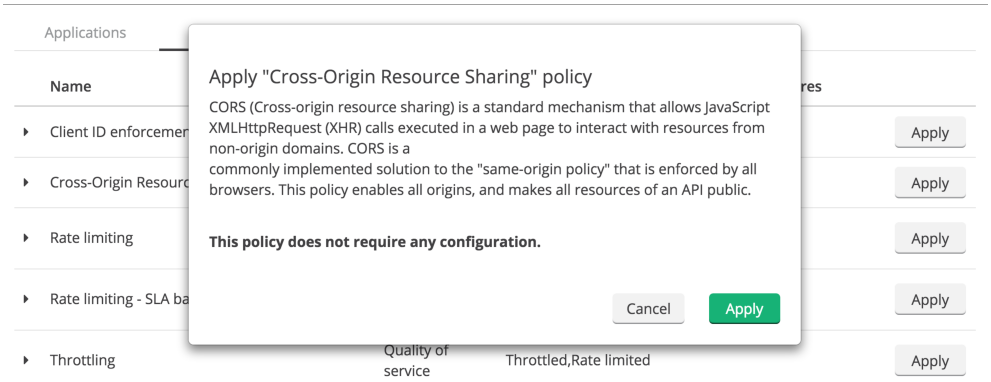
CORS lets you enable these cross-domain calls, while also letting you specify host limitations (for example, only calls from x-host will be allowed; all other hosts will be denied) using the Access-Control-Allow-Origin header. The idea is that this will help reduce the security risks by only letting certain hosts take advantage of this functionality.

With that said, simply restricting hosts does not ensure that your application will be safe from JavaScript attacks, as clients can easily manipulate the JavaScript code themselves using freely available browser-based tools (such as Firebug or Inspector).

Being client-side, CORS can also present other security risks. For example, if a developer chooses to control your API via client-based JavaScript instead of using a server-side language, they may expose their API key, access tokens and other secret information.

As such, it's important to understand how users will be utilizing CORS to access your API to ensure that secret or detrimental information is not leaked.

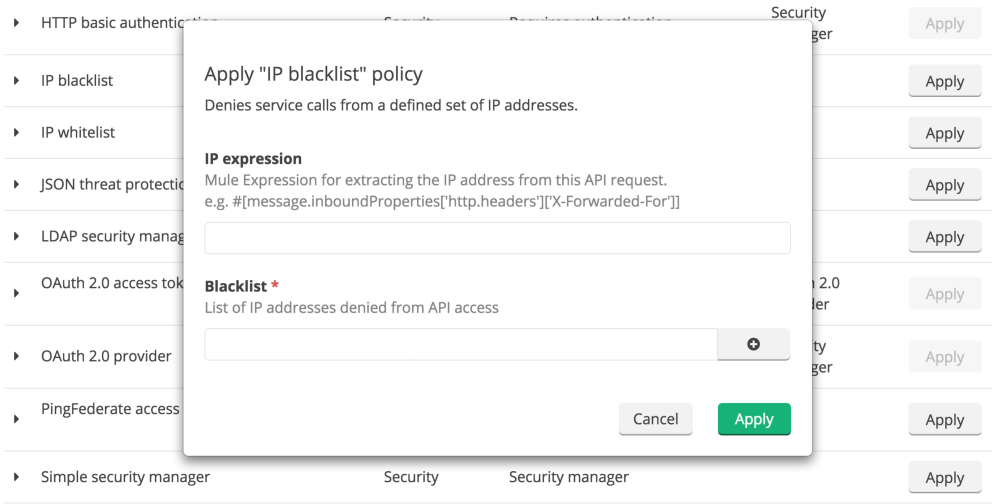
Keep in mind that every API manager operates differently. Some API managers may let you set up a list of allowed hosts, while others may issue a blanket statement allowing *every* host to make cross-origin requests.



As such, the golden rule for CORS, unless you have a specific-use case, is to leave it disabled until you have a valid, well-thought-out reason for enabling it.

IP Whitelisting/Blacklisting

While IP whitelisting/blacklisting is not an acceptable method for securing an API by itself (as IP addresses can be spoofed and hackers often times use multiple addresses), it does help provide an added layer of security to prevent or allow known IPs based on previous interactions. For example, you can choose to whitelist your or your partner's dedicated IPs, or blacklist IPs that have a history of making malicious calls against your server or API. You can also block IPs from users that have abused your API to prevent their server from communicating with it (something they could easily do if you have a free tier by simply creating a new account with a different email).



As we look at IP whitelisting/blacklisting, it's important to remember that security functions in layers. For example, if you think of a prison, it's important to have a multitude of security clearances so that if by chance someone bypasses the first layer, they will be stopped at the second or third. In the same way, we need to layer our security processes for our API, both on the API management/proxy side, and within our architecture.

IP whitelisting/blacklisting fits very nicely into this layered system and can help kill some calls before running any additional checks on them. But again, it's important to remember that IP addresses may change and can be spoofed, meaning you may not always be allowing "good" users, and you may not always be stopping malicious traffic. However, if you are selective in this process, the chances that you will disrupt well-intentioned users from interacting with your API are pretty slim.

XML Threat Protection

With the growth and focus of SOA in enterprise architectures, hackers worked to find a new way to exploit vulnerabilities, often times by injecting malicious code into the data being passed through the system. In the case

of XML services, users with malicious intent could build the XML data in such a way as to exhaust server memory, hijack resources, brute force passwords, perform data tampering, inject SQL, or even embed malicious files or code.

In order to exhaust server memory, these attackers might create large and recursive payloads, something that you can help prevent by limiting the length of the XML object in your API manager, as well as how deep into the levels the XML may go.

Along with memory attacks, malicious hackers may also try to push through malicious XPath/XSLT or SQL injections in attempt to get the API layer to pass along more details than desired to services or the database.

Malicious attacks may also include embedding system commands in the XML Object by using CDATA or including malicious files within the XML payload.

Of course, there are several more XML-based attacks that can be utilized to wreak havoc on an API and the underlying architecture, which is why having XML threat protection in place is key to ensuring the safety of your API, your application and your users. Again, since security is built in layers, while the API manager can help prevent some of these threats, monitor for malicious code and limit the size of the XML payload or the depth it can travel, you will still want to be meticulous in building your services architecture to ensure that you are eliminating threats like SQL and code injection on the off chance they are missed by your API gateway.

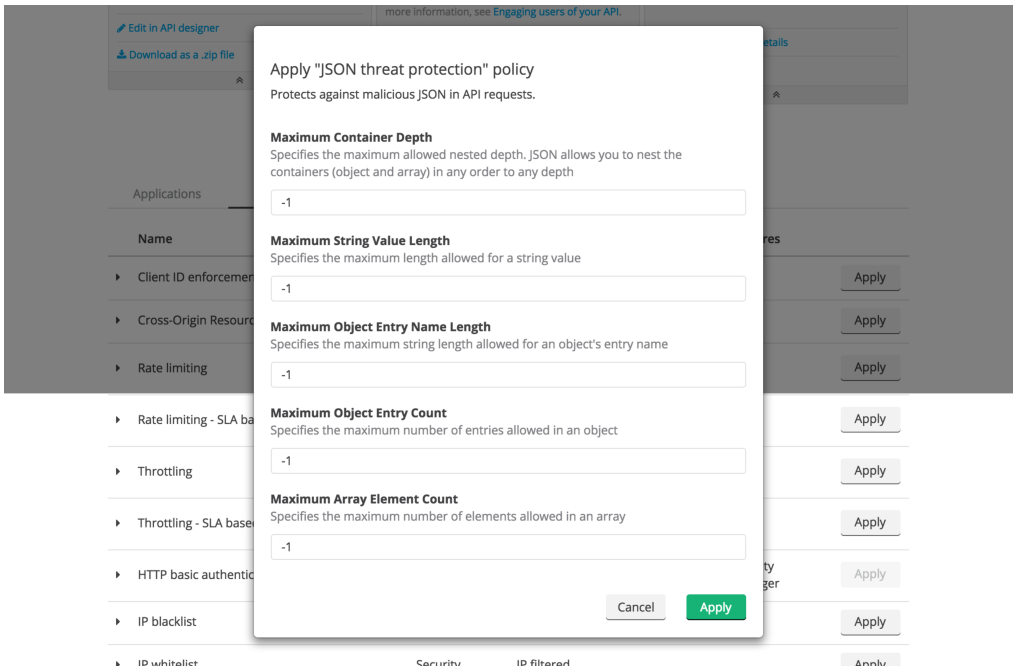
JSON Threat Protection

Similar to XML, JSON falls victim to several of the same malicious attacks. Attackers can easily bloat the JSON and add recursive levels to tie up

memory, as well as inject malicious code or SQL that they anticipate your application will run.

As with XML threat protection, you will want to limit the size and depth of the JSON payload, as well as constantly be on the watch for security risks that might make it through the API gateway including SQL injections and malicious code the user wants you to evaluate.

MuleSoft's API Manager lets you set up custom policies to help prevent XML and JSON Threat Protections:



12

Documenting & Sharing Your API

Since the goal of any API is developer implementation, it's vital not to forget one of your most important assets—one that you should be referencing both in error messages and possibly even in your hypermedia responses: documentation.

Documentation is one of the most important factors in determining an API's success, as strong, easy-to-understand documentation makes API implementation a breeze, while confusing, out-of-sync, incomplete or convoluted documentation makes for an unwelcome adventure—one that usually leads to frustrated developers utilizing competitor's solutions.

Unfortunately, documentation can be one of the greatest challenges, as up until now we have been required to write the documentation as we go, try to pull it from code comments or put it together after the API has been developed.

The challenge is that not only should your documentation be consistent in its appearance, but also consistent with the functionality of your API and in sync with the latest changes. Your documentation should also be easily understood and written for developers (typically by an experienced documentation team).

Until recently, solutions for documentation have included expensive third-party systems, the use of the existing CMS (Content Management System), or even dedicated CMS's based on open source software such as Drupal/WordPress.

The challenge is that while expensive API documentation-specific solutions may provide consistency regarding the look and feel of your API (something harder to maintain with a CMS), they still rely on the manual effort of the developer (if derived from the code) or a documentation team to keep them in sync.

However, with the expansion of open specs such as RAML—and the communities surrounding them—documentation has become incredibly easier. Instead of trying to parse code comments and have inline descriptions written (usually) by developers, the documentation team is still able to provide descriptive documentation in the spec, and all code parameters/examples are already included, making the transition to documentation a snap.

And with the explosion of API documentation software-as-a-service (SaaS) companies that utilize and expand on these specs, creating an effective API portal and documentation has never been easier or less expensive.

However, before we jump into the different documentation tools, it's important to understand what makes for good documentation.

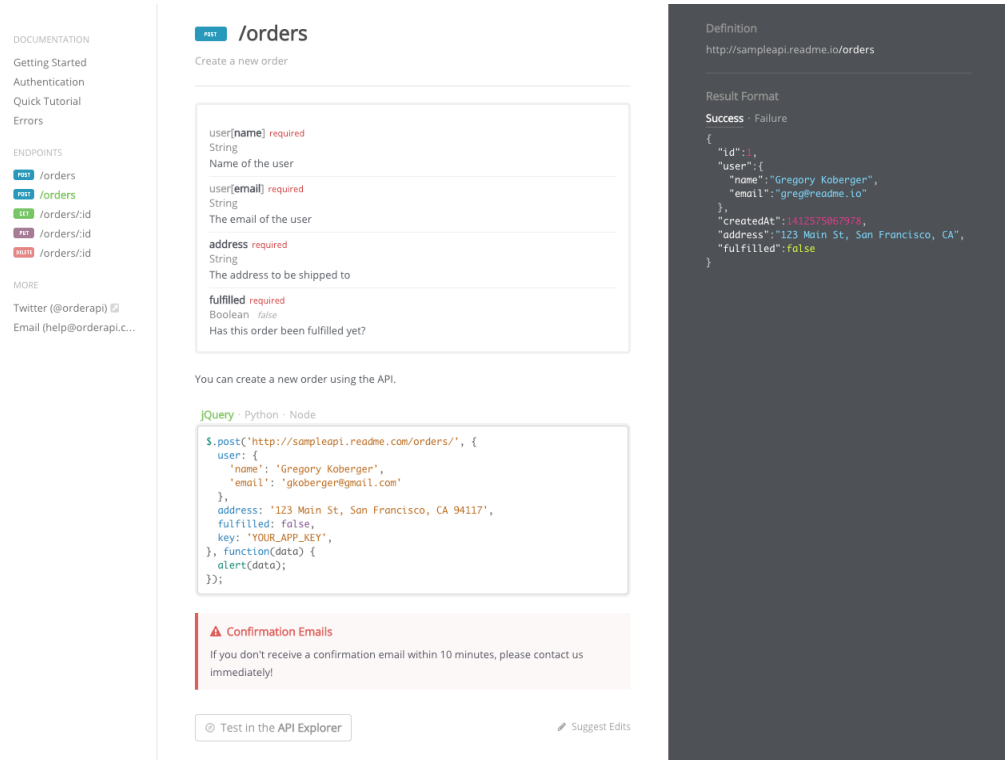
Writing Good Documentation

Good documentation should act as both a reference and an educator, letting developers quickly obtain the information they are looking for at a glance, while also reading through the documentation to glean an understanding of how to integrate the resource/method they are looking at.

As such, good documentation should be clear and concise, but also visual, providing the following:

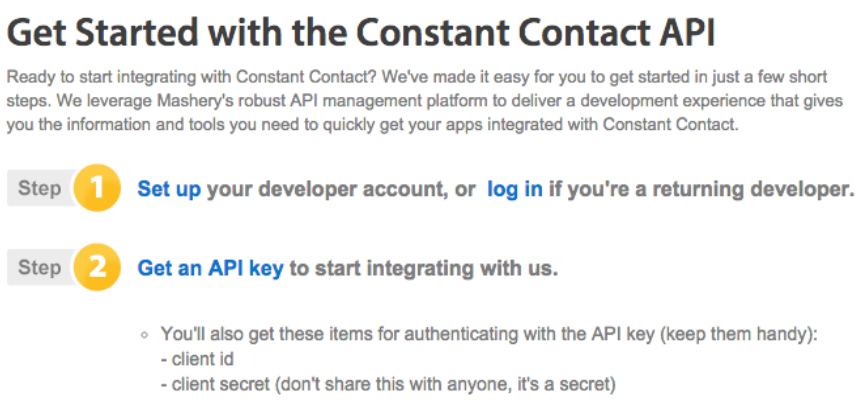
- A clear explanation of what the method/resource does
- Call outs that share important information with developers, including warnings and errors
- A sample call with the correlating media type body
- A list of parameters used on this resource/method, as well as their types, special formatting, rules and whether or not they are required
- A sample response, including media type body
- Code examples for multiple languages including all necessary code (e.g. Curl with PHP, as well as examples for Java, .Net, Ruby, etc.)
- SDK examples (if SDKs are provided) showing how to access the resource/method utilizing the SDK for their language
- Interactive experiences to try/test API calls (API Console, API Notebook)
- Frequently asked questions/scenarios with code examples
- Links to additional resources (other examples, blogs, etc.)
- A comments section where users can share/discuss code
- Other support resources (forums, contact forms, etc.)

One such service that does an excellent job regarding documentation as a whole is ReadMe.io, a SaaS-based API documentation company.



Of course, the ReadMe.io example isn't perfect, but their interface does let you add additional items, and overall it gives you an example of a visual interface that developers can access to quickly obtain information about the type of call they are trying to make while also having access to code examples and being alerted to potential issues.

Another good example of API documentation is Constant Contact's setup, which provides chronological steps for developers to aid them through the onboarding process.



Get Started with the Constant Contact API

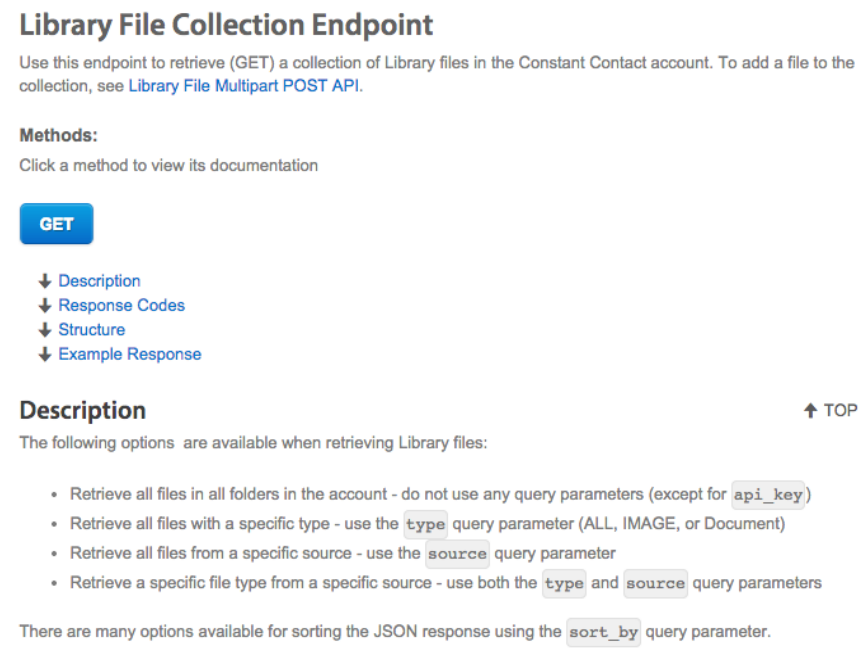
Ready to start integrating with Constant Contact? We've made it easy for you to get started in just a few short steps. We leverage Mashery's robust API management platform to deliver a development experience that gives you the information and tools you need to quickly get your apps integrated with Constant Contact.

Step 1 **Set up your developer account, or log in if you're a returning developer.**

Step 2 **Get an API key to start integrating with us.**

- You'll also get these items for authenticating with the API key (keep them handy):
  - client id
  - client secret (don't share this with anyone, it's a secret)

Constant Contact also sections their documentation, providing links to the different sections at the top so that developers can quickly scroll to the information they are looking for, as well as giving a strong overview and description of the resource as a whole.



Library File Collection Endpoint

Use this endpoint to retrieve (GET) a collection of Library files in the Constant Contact account. To add a file to the collection, see Library File Multipart POST API.

Methods:

Click a method to view its documentation

GET

- Description
- Response Codes
- Structure
- Example Response

Description

TOP

The following options are available when retrieving Library files:

- Retrieve all files in all folders in the account - do not use any query parameters (except for `api_key`)
- Retrieve all files with a specific type - use the `type` query parameter (ALL, IMAGE, or Document)
- Retrieve all files from a specific source - use the `source` query parameter
- Retrieve a specific file type from a specific source - use both the `type` and `source` query parameters

There are many options available for sorting the JSON response using the `sort_by` query parameter.

They also provide a nice, clean interface for describing the query parameters:

GET: https://api.constantcontact.com/v2/library/files

Test API

name	type	default	description
api_key	query		REQUIRED; The API key for the application
limit	query	50	Specifies the number of results displayed per page of output, from 1 - 1000, default = 50. See Paginated Output for more information on using `limit`.
sort_by	query	CREATED_DATE_DESC	Specifies how the list of files is sorted; valid sort options are: • CREATED_DATE - sorts files by date created, ascending (earliest to latest) • CREATED_DATE_DESC - (default) sorts files by date created, descending (latest to earliest) • MODIFIED_DATE - sorts by date the file was last modified, ascending (earliest to latest) • MODIFIED_DATE_DESC - sorts by date the file was last modified, descending (latest to earliest) • NAME - sorts files alphabetically by name, a to z • NAME_DESC - sorts files alphabetically by name, z to a • SIZE - sorts by file size, smallest to largest • SIZE_DESC - sort by file size, largest to smallest • DIMENSION - sorts by file dimensions (hxw), smallest to largest • DIMENSION_DESC - sorts by file dimensions (hxw), largest to smallest

The available response codes:

Response Codes

↑ TOP

code	description
200	Request was successful
401	Authentication failure
406	Unsupported Accept Header value, must be application/json
500	Internal server error occurred

And the overall structure of the API response:

Structure

↑ TOP

property	type (max length)	description
created_date	string	Date the file was added to MyLibrary, in ISO-8601 format
description	string (100)	Description of the file provided by user
file_type	string	Specifies the file type, valid values are: JPEG, JPG, GIF, PDF, PNG
folder	string (80)	Name of the folder the file is in
folder_id	string	Unique ID of the folder the file is in

As well as an example response to inform their developers of what they should expect to receive back, letting developers prepare their script to handle the data and compare what they *are* getting back to what they *should be* getting back:

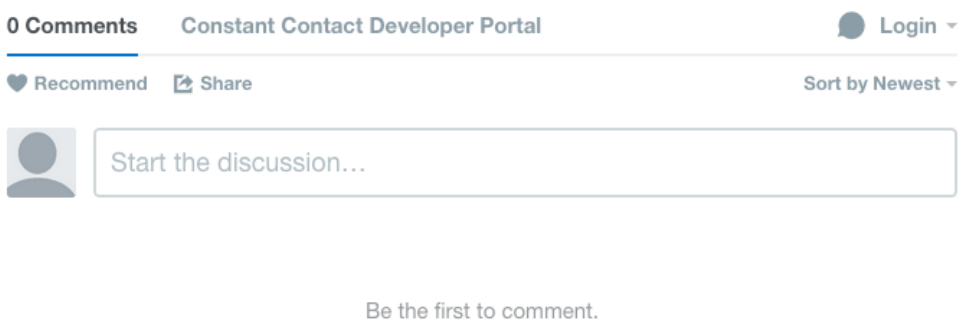
Example Response

↑ TOP

```
{
    "meta": {
        "pagination": {}
    },
    "results": [
        {
            "name": "IMG_0341.JPG",
            "id": "9",
            "description": "buildings_steel",
            "folder": "wildflowers",
            "height": 612,
            "width": 792,
            "size": 304560,
            "url":
"https://origin.ih.l1.constantcontact.com/fs134/1100394770946/img/9.jpg",
            "source": "MyComputer",
            "status": "Active",
            "thumbnail": {
                "url":
```

This format is extremely user-friendly and helps developers get up and running with the Constant Contact API quickly. It also has the added benefit of a “Test API” button located at the top of each section, letting developers read about the resource/method and then quickly try it out in the interactive console.

Last but not least, Constant Contact integrates comments into their API documentation through the use of Disqus, a simple and free comment management tool that only requires a snippet of code to be placed on your site.



However, you’ll notice that the Constant Contact documentation, as well written and thorough as it is (kudos to Richard Marcucella), lacks code examples—something that was a constant pain point for getting developers started with the resource while I was working as a Developer Evangelist at Constant Contact.

For many developers, code examples provide the key to integrating with your API, allowing them to quickly copy and paste it in without having to spend the time to “learn your API.” While not necessarily ideal, this is a reality of how a large portion of programming works today, and something developers will be looking for on your site.

Twilio, a communications company built as an API provider, does a pretty good job in the code example department, providing code examples for

eight different languages based on their SDK. This means that if you've installed the Twilio SDK, integrating with their service takes only minutes, as you can literally paste in the code example, only having to replace the placeholders with your own values.

Twilio has also mastered the "five-minute demo" in regards to utilizing their API. As you build out your documentation, this five-minute window should be your goal. From reading your documentation, developers should be able to create a working example (even if it's super basic) in five minutes or less. If it takes longer than five minutes, you may need to evaluate whether or not your documentation is as clear and concise as it should be.

Twilio's documentation with code examples:

Resource URI

```
/2010-04-01/Accounts/{AccountSid}/Queues/{QueueSid}
```

Resource Properties

A Queue resource is represented by the following properties:

Property	Description
`Sid`	A 34 character string that uniquely identifies this queue.
`FriendlyName`	A user-provided string that identifies this queue.
`CurrentSize`	The count of calls currently in the queue.
`MaxSize`	The upper limit of calls allowed to be in the queue. The default is 100. The maximum is 1000.
`AverageWaitTime`	The average wait time of the members of this queue in seconds. This is calculated at the time of the request.

HTTP GET

Returns a representation of the Queue identified by {QueueSid}.

Example

JSON | XML | PHP | Python | C# | Java | Ruby | Node.js

```php
<?php
// Get the PHP helper library from twilio.com/docs/php/install
require_once('/path/to/twilio-php/Services/Twilio.php'); // Loads the library

// Your Account Sid and Auth Token from twilio.com/user/account
$sid = "AC5ef8732a3c49700934481addd5ce1659";
$token = "{{ auth_token }}";
$client = new Services_Twilio($sid, $token);

// Get an object from its sid. If you do not have a sid,
// check out the list resource examples on this page
$queue = $client->account->queues->get("QU5ef8732a3c49700934481addd5ce1659");
echo $queue->average_wait_time;
```

Documentation Tools

One of the key benefits to Spec-Driven Development is that instead of having to rely on costly and complex vendors—or trying to build an ad-hoc API documentation solution out of a CMS like WordPress or Drupal—specs like RAML, Swagger and API Blueprint have strong open source communities surrounding them that offer prebuilt documentation tools you can use.

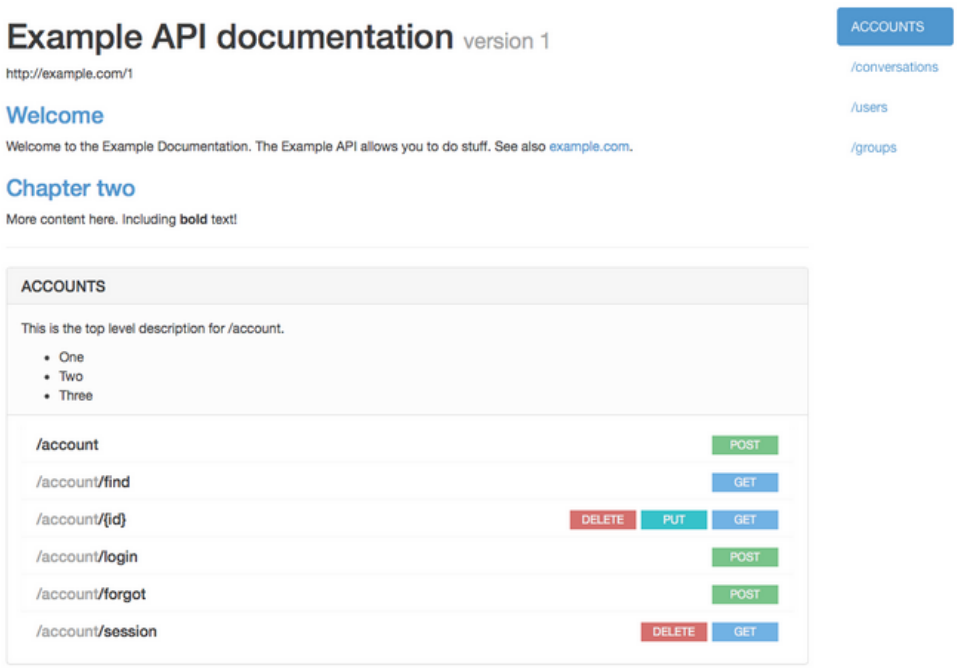
While each offers its own unique toolset, and each spec has its own strengths and weaknesses (refer to Chapter 3), since we've been using RAML so far, we'll focus on the tools available from the RAML community.

The RAML community has already put together parsers for several different languages including Java, Ruby, PHP and Node, as well as full scripts to manage API documentation while providing interactive environments such as the API Console and API Notebook. These tools help you provide documentation as shown in the ReadMe.io, Constant Contact and Twilio examples above with little to no work on your part (other than the installation and, of course, defining your RAML).

The following tools are available at http://raml.org/projects and can be downloaded freely.

RAML to HTML

RAML to HTML is a Node.js script that generates documentation into a singular HTML page from the RAML spec. This provides a quick overview of your code, in a similar format as Swagger's console, letting developers view and try out your API.



The RAML to HTML script also has third-party plugins for both Gulp and Grunt.

RAML to Markdown

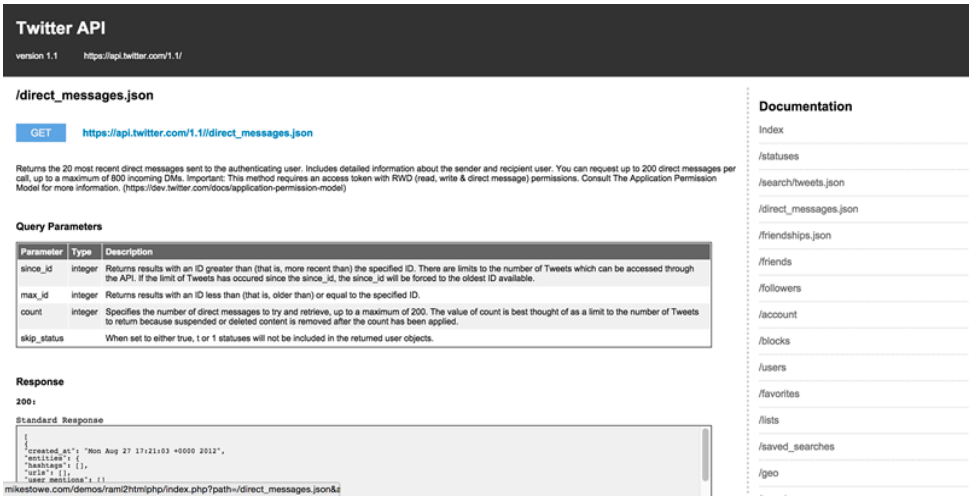
In the event that you are using a CMS that takes advantage of markdown, you can use this Node.js script to quickly transform your RAML spec in markdown that can be copied and pasted in your markdown CMS of choice.

RAML to Wiki

This Node.js script is designed to let you easily port your RAML to Atlassian's Confluence or Jira style markdown, letting you integrate documentation into your external or internal wikis, as well as bug tracking systems.

RAML 2 HTML for PHP

Based on RAML to HTML, RAML to HTML for PHP renders your RAML in real time, creating a full-fledged set of documentation that developers can navigate, with unique pages for each resource and method. HTML to RAML for PHP is also based on a templated system, letting you customize the look and feel of your development portal and even add/remove different aspects of documentation.



While these tools may not encompass all of the items that you are looking for in documentation (e.g. code examples, call outs, comments, etc.) they do provide a good starting point. And in the case of RAML 2 HTML for PHP, they provide a toolset that lets you easily integrate these features into your documentation.

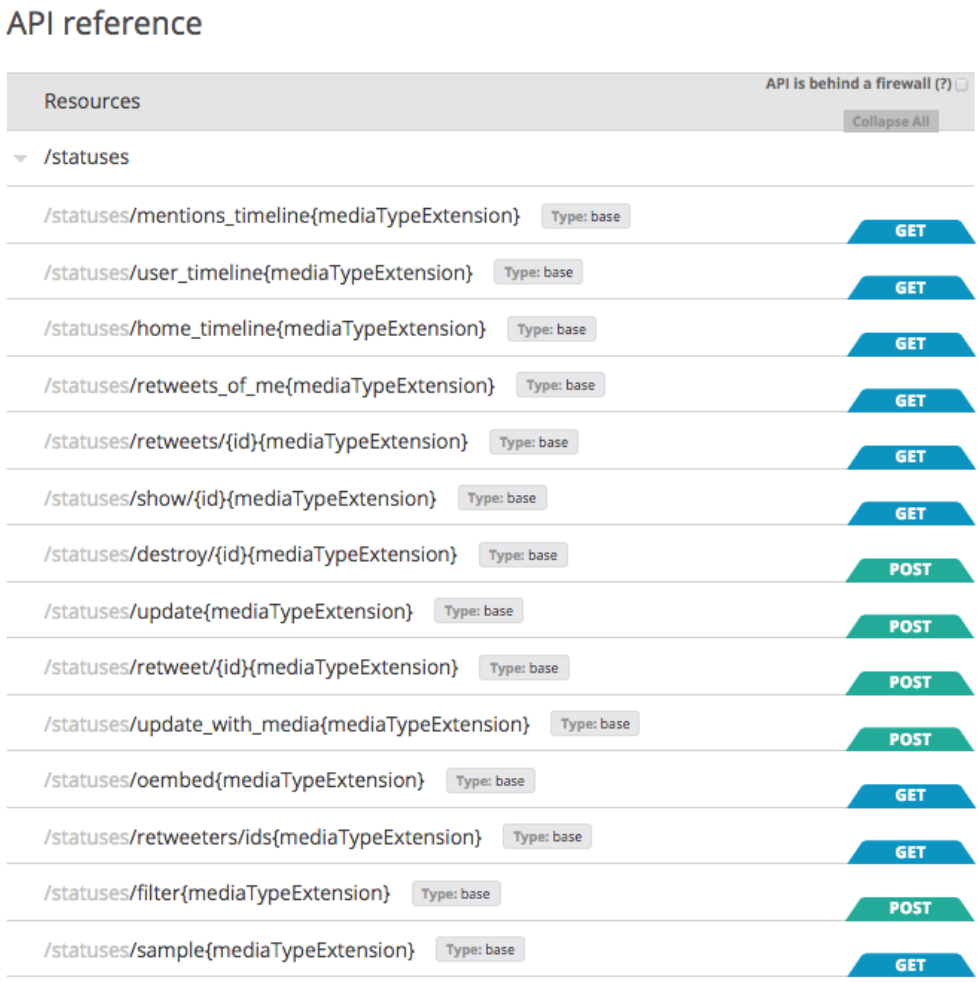
As mentioned, there are also a growing number of third-party SaaS API documentation services such as ReadMe.io that let you pull in your spec to build out your API documentation. The trick is you'll want to be sure that whatever solution you choose can pull in any updates to your spec without overwriting the additional data you've provided (e.g. callouts) and that everything you are documenting is in fact documented in your spec (to prevent this from happening).

Along with providing textual documentation, RAML also provides two powerful, interactive tools to help developers interact with, and better understand, your API. The first is the API Console.

API Console

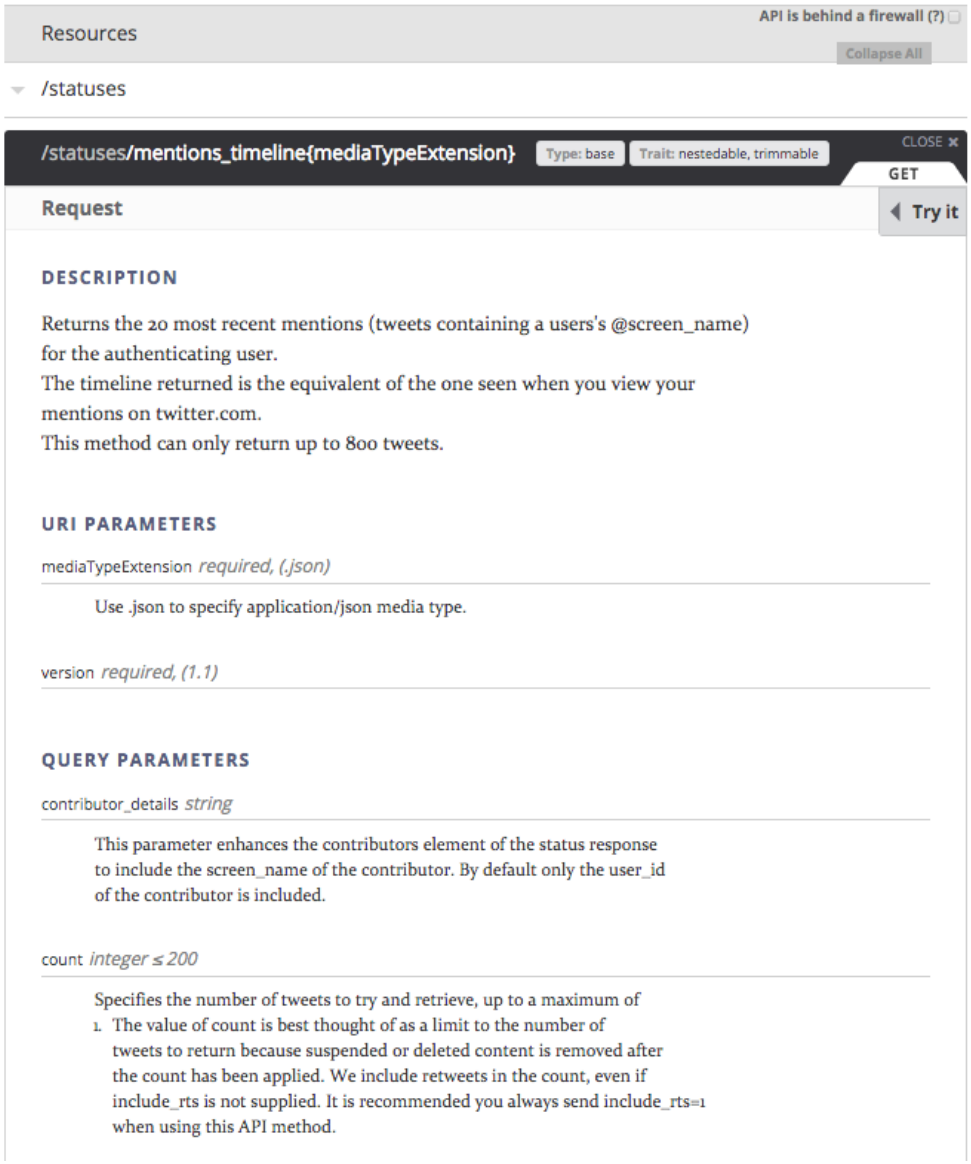
The API Console is available both as an open source project (on the RAML website) and as a freely hosted service from MuleSoft (anypoint.mulesoft.com/apiplatform). The API Console provides a unique way of documenting, demonstrating and letting developers interact with your API.

In the latest version, documentation plays an even heavier role, letting developers quickly view the different endpoints available and then—when drilling down into a resource—providing complete and detailed documentation on the resource definition.

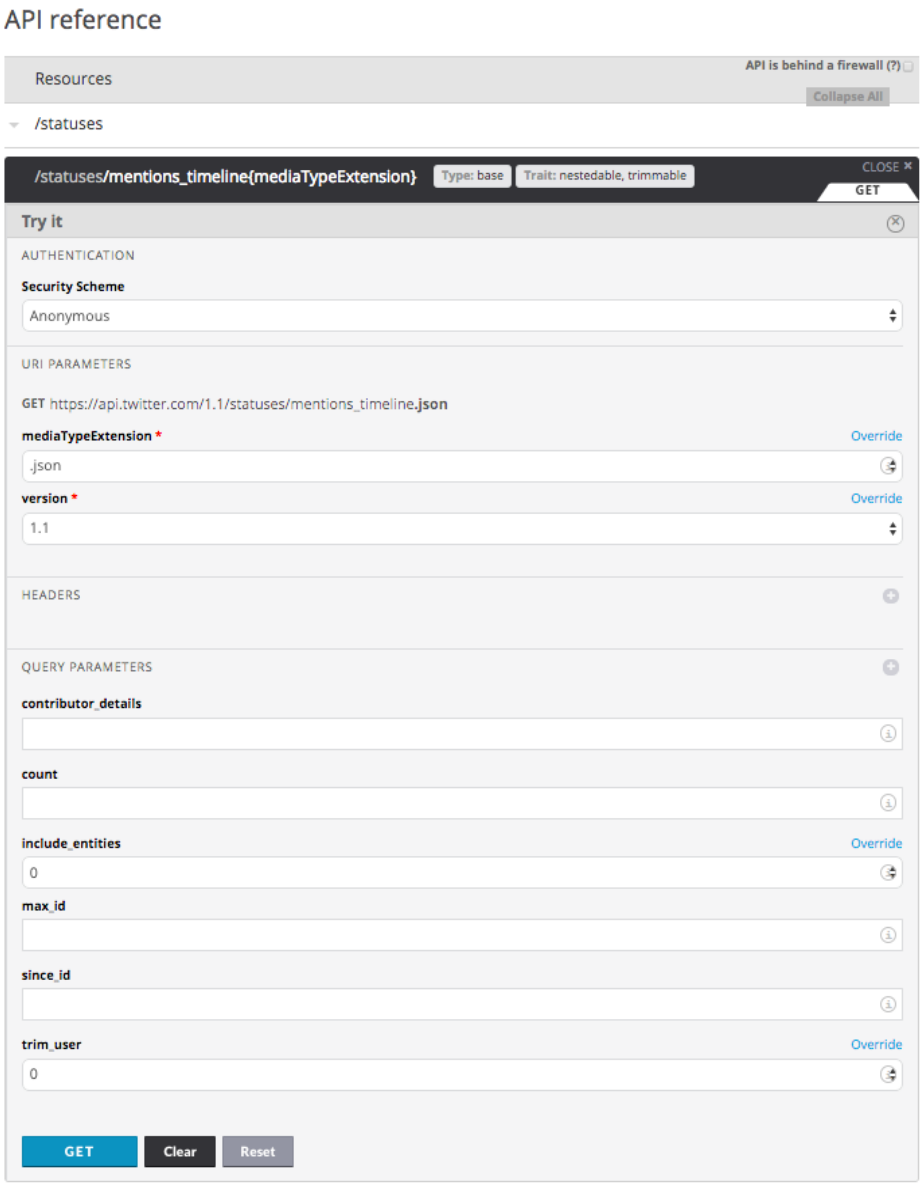


Once the user selects the method of the resource they are wanting to learn more about, they are presented with a reference panel for that method, including the RAML provided description, parameters, responses, and more:

API reference



As with the Constant Contact API documentation, there is a “Try It” button that pulls up an interactive console that lets the developer enter the necessary authentication and parameters to make an actual API call.



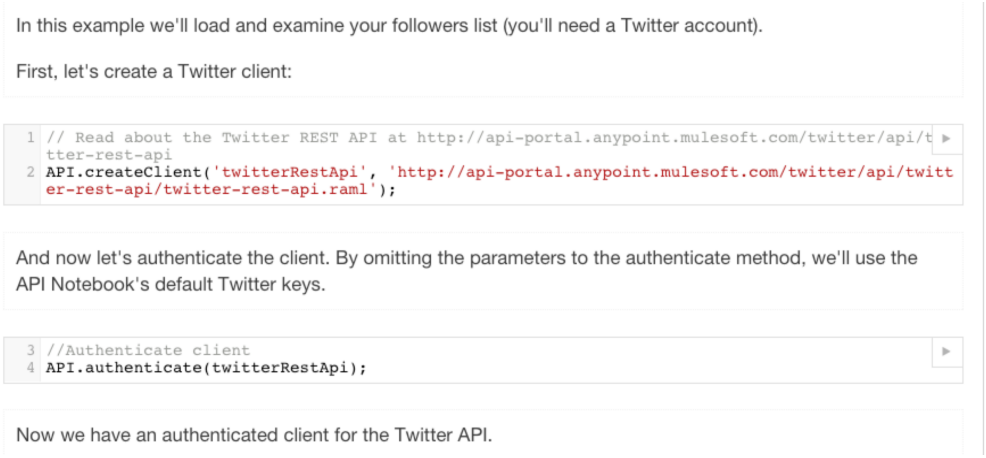
The API Console provides you with a quick way to share your documentation with developers, while also providing a powerful tool for testing out API calls and debugging calls when something goes wrong in their code.

But what the API Console doesn't do is provide a way for you to set up example use-case scenarios for your API, or let developers provide feedback/their use cases without sharing proprietary code. This next tool, the API Notebook provides an intuitive way to highlight your API's abilities, let developers explore your API and garner feedback.

API Notebook

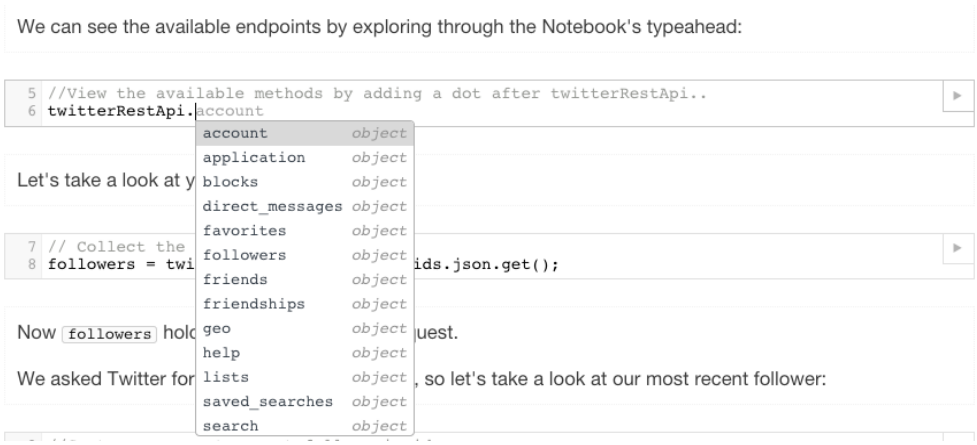
Another freely available JavaScript based tool, the API Notebook lets you create scenarios/examples of API usage for your developers while walking them through what each step does.

For example, in the screenshot below, we will first create an API client by grabbing the Twitter RAML URL, and then we will authenticate into Twitter using OAuth (as described in the RAML file):



To run this notebook, we would simply click “Play” on the step(s) we want to run (for example, if we click “Play” for the API.authenticate step, it would run all previous steps), or we can click the “Play All” button (not shown) to run all the steps.

Once we have created an API client, we now have the freedom to explore that client by simply adding a “.” to the API alias (in this case twitterRestApi).



This means that after reviewing your example, your developers can branch off and create their own, exploring your API and the resources available to them without having to read any documentation.

As each call is made, the response from the server is made available to the developer, letting them view the headers as well as the body response.

Along with acting as a useful tool by letting developers try out scenarios you have created, developers can also copy or create their own notebooks, sending information back to you. In the case of support, this lets a developer show you what they are trying to do with your API and what errors they are getting without revealing proprietary code. This also means your support team can see first-hand exactly what the issue is in a way that is reproducible.

The API Notebook also lets you integrate any other API defined with RAML into your calls, letting developers try your API in conjunction with other APIs they may be using, as well as letting them manipulate and modify the output in the notebook using JavaScript.

For example, going back to the sample Twitter API Notebook, we could determine who our last follower was by adding in the following JavaScript code:

Let's take a look at your followers:

```
// Collect the IDs of your followers
followers = twitterRestApi.followers.ids.json.get();
```

Now `followers` holds the response to our request.

We asked Twitter for a list of our followers' ids, so let's take a look at our most recent follower:

```
//Capture our most recent follower's id
lastFollowerId = followers.body.ids[0];
'Our most recent follower ID is ' + lastFollowerId;
```

Great! Let's get some information about this follower.

```
// Show profile information.
lastFollower = twitterRestApi.users.show.json.get({
  user_id: lastFollowerId
});
```

```
// Display screen name.
'Your last follower\'s screenname is ' + lastFollower.body.screen_name;
```

The API Notebook provides a whole new way for developers to interact with, test and learn about your API, while also making the job of support that much easier. The ability to provide explorative scenarios for your developers, while also letting developers create their own scenarios that can be shared with you or others using markdown, make the API Notebook possibly one of the most powerful API exploration tools on the Internet today.

The API Notebook is currently only available for RAML, and again, is freely available on the RAML website (raml.org/projects). The API Notebook is also available as part of MuleSoft's API Portal (a tool for building your own API Portals – anypoint.mulesoft.com/apiplatform) or as a universally accessible tool at apinotebook.com.

Support Communities

Along with providing documentation, you'll want to make sure that your developers have a support community where they can interact with other developers to get the answers to their questions. This may be a dedicated support channel (either through email or a support ticket system), or an unmanned support community, such as a forum/redirection to support sites such as StackOverflow.

When evaluating how to set up your support community, it's important to understand what your resources and objectives are, as well as the community that you'll be serving. For an internal API, this may be something as simple as having the API development team handle support tickets/emails.

For a public API, you may want to consider setting up a forum as a primary source for answering questions and directing people to the forum if they are unable to find the answer in your documentation or an FAQ. Even if you choose to offer free, dedicated support to your developers, by creating and hosting a forum you enable the community to share ideas, become more engaged and reduce the number of support tickets. However, you'll need to carefully manage and moderate your forum, being careful to eliminate spam/trolling without affecting the free speech of your developers.

Forums also present a way for you to hear the complaints of your developers, as well as get broad feedback on different ideas and implementations. Some companies have elected to include a voting system

that lets users vote up forum questions and replies in a similar manner to StackOverflow, while also creating a separate board where developers can up vote feature requests and ideas (helping you gauge the demand of these items).

Some companies have elected to do away with support altogether, instead pushing developers to third-party services such as StackOverflow. This is perfectly acceptable, however, you must carefully weigh the pros and cons before doing it, as it may adversely affect your community, your ability to solicit feedback, developer usage and, ultimately, your company if questions on these third-party sites go unanswered. Regardless of whether they are your primary source of support, you should still have your team monitor these services when hosting public/open APIs to ensure that developers are able to get the answers they need, while also highlighting pain-points in terms of integrating your API.

Keep in mind, if you are looking at building out a Developer Relations program around your API, the first step is to create a community of support that brings developers in contact with your company. By having this community already in place, your Developer Relations team will be able to converse with these developers, and you will already have advocates who are excited to talk about and share your API with their friends, co-workers and peers.

All Developer Relations programs start with a focus on community.

SDKs and Client Libraries

Software Development Kits, or SDKs, present an interesting paradox in terms of developer onramp and support. On one hand they can greatly reduce the onramp, as developers can simply drop in the SDK and begin working with your API, using just the code or aspects that they need. On the other hand, this requires the developer to update the SDK when

changes are made to it in order to access newer aspects of the API, and also requires you to maintain and support them.

The biggest advantage of an SDK is that it doesn't require the developer to learn your API and understand the rules of REST. Instead, all they see is code—not necessarily resources or even methods. They do not need to understand the difference between POST, PUT and PATCH—all they need to do is add some code, and it works.

But again, on the flip side, complex SDKs require developers to learn a new set of code, even if they already understand how to interact with your API.

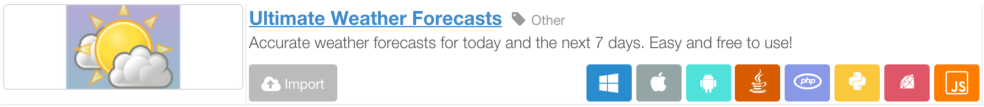
Perhaps more than full-fledged SDKs I am a fan of code wrappers, or intuitive interfaces that take away the call logic (e.g. CURL for PHP). These tend to be more lightweight, with less dependencies, and are far more flexible and forgiving. However, they require that the developer have at least a partial understanding of your API.

Regardless of your decision, if there are code libraries/ SDKs out there on your public API, your company will need to support them. Perhaps not in regards to contributing code, but from a "how to use it" standpoint, as developers will be asking about this.

If you choose to provide your own SDKs, keep in mind that you will want to provide ones for the most popular languages your clients are using, and this could mean—for example, in the case of Twilio—supporting eight different languages and having eight different SDKs. As you build out the SDKs, you want developers who are proficient in each language producing them, which creates a contracting/outsourcing nightmare for most companies.

Because SDKs are so popular, but also a pain for companies to build and maintain, we've seen a couple businesses pop up with the solution to this

problem as their sole focus. Companies like APIMATIC.io and REST United create SDKs built on demand to your specification to meet your client's needs. For example, APIMATIC.io currently generates SDKs for Android, Windows, iOS, Java, PHP, Python, Ruby and Angular JS, all with the press of a button. They also provide a widget that allows developers to download the SDK directly from your site instead of having to visit the APIMATIC.io directory.



REST United (restunited.com) goes a step further, having both documentation with code samples and SDKs for Android, C#, ActionScript, Java, Objective-C, PHP, Python, Ruby and Scala.

Of course, one challenge when using generated SDKs is quality, and both of these companies are still relatively new and still testing the power of their solutions. But this may be a good starting point, as they have the ability to generate a bulk amount of the code and then clean it up, making your SDK easier both to build and maintain.

The simple reality of an SDK or client library is this—it can be an effective and very helpful tool for your users, but in the process of solving one problem—if you're not careful—you may end up creating more. Simply having an SDK isn't enough; you also need to form a plan for how the SDK will be maintained (whether by you or the community) and how it will be supported (since again, regardless of your official support policy, people will ask your support teams how to integrate your API using both the SDK and other technology).

In an early presentation I ended with the following quote: “SDKs save lives.” The truth is, I think *code samples* save lives. And regardless of your SDK decision, code samples will be integral to the success of developers when trying to integrate your API.

13

A Final Thought

One of the challenges of writing a "generalized" API book is just that. It becomes generalized—a jack-of-all-trades but master of none. There is so much more that could have been covered in every chapter if we had the time and space to cover it. Of course that would also require you finding the time and space to read it. Each chapter should be looked at as a starting point, not a complete reference, as it only touches the tip of the iceberg. But hopefully this book has provided you with a strong base on which to build the design of your API.

But as you proceed with your API, I want to share just one more lesson life has taught me. Oftentimes, especially as developers, we want to push the envelope and create new things. But an API that will be relied upon by your company and—potentially—third parties, is neither the right time nor place to try new things and travel through uncharted waters. Instead, it is the time to apply the best practices that we know work—the practices we know will ensure a long and healthy API life.

That doesn't mean that one shouldn't innovate. In fact, I believe quite the opposite. I can't encourage innovation enough and strongly believe that this

is one of the greatest benefits the open source community gives us—the chance to innovate and the chance to validate our creations. Continue to innovate and try new things, but test your thoughts and processes in personal projects and get community validation before trying them in a production-level environment. Remember, pushing an API to production is like squeezing toothpaste out of the tube—it's really easy to do, but extremely difficult to take back.

Growing up, my father would constantly remind me of the KISS principle: "Keep it simple, stupid." It became a running joke in my family, but this principle applies to APIs perhaps more than anything else, since when we are creating an API, we are constructing the foundation on which we will need to continue building.

By keeping it simple, we help make our API easier to use, easier to maintain and easier to extend. Avoid the urge to go wild—to create "amazing" and "shiny" interactions. Because as enticing as they may be, in the end, the sole purpose of your API is to let developers interact with your core application's services and data, and to make doing so as easy, efficient and simple as possible.

Again, hopefully this book helps guide you towards best practices, highlights problems to watch out for, and provides proven methods to help you build your API in the strongest and most enduring way possible. By using Spec-Driven Development, you not only create a contract, but also test the contract before investing time, and then enforce it—not just on the client's end, but also your own.

If I had to highlight the key takeaways for building a successful API, I would say remember these rules of thumb:

By using tools like RAML, you allow yourself to save time and energy, aggregating different processes into a singular control and allowing you to

expand upon the tools you offer your developers while reducing your own workload.

By incorporating best practices you create an API that is easily recognizable and consistent, helping developers take advantage of your offerings with a reduced learning curve while also making it easier on your support and maintenance teams.

By utilizing an API Manager you provide yourself the reassurance that your API has built-in user management, provisioning, scalability and security.

And by providing complete, easy-to-understand documentation, you are ensuring that when your developers do have questions, they can quickly obtain the information without having to spend hours searching or contact support for answers.

With all of these steps, and by remembering the KISS principle, you can be on your way to building the API of your dreams. And if you choose to follow the REST constraints, you will have one that provides both you—and your developers—with peace of mind and undisturbed REST.

Appendix: More API Resources

API Courses (live training)

MuleSoft – http://training.mulesoft.com

API Directories

ProgrammableWeb – http://www.programmableweb.com

PublicAPIs – http://www.publicapis.com/

APIs.io – http://www.apis.io

API Tutorials

REST API Tutorial – http://www.restapitutorial.com/

API Definition Specs

RAML – http://raml.org

Swagger – http://www.swagger.io

API Blueprint – http://apiblueprint.org

API Hypertext Link Specs

HAL – http://stateless.co/hal_specification.html

JSON API – http://jsonapi.org/

JSON-LD – http://json-ld.org/

Collection+JSON – http://amundsen.com/media-types/collection/

CPHL – https://github.com/mikestowe/CPHL/blob/master/README.md

Siren – https://github.com/kevinswiber/siren/blob/master/README.md

Uber – http://rawgit.com/uber-hypermedia/specification/master/uber-hypermedia.html

Yahapi - https://github.com/Yahapi/yahapi/blob/master/pages/home.md

API Descriptive Error Formats

JSON API – http://jsonapi.org/format/#errors

Google Errors – http://bit.ly/1wUGinJ

vnd.error – https://github.com/blongden/vnd.error

API Mocking Services

MuleSoft – https://anypoint.mulesoft.com/apiplatform/

Apiary – http://apiary.io/

Mockaeble – http://mockable.io

API Frameworks

Apigility – https://www.apigility.org/

Grape – http://intridea.github.io/grape/

Rails API – https://github.com/rails-api/rails-api

Restify – http://mcavage.me/node-restify/

Jersey – https://jersey.java.net/

API Testing

API Science – https://www.apiscience.com/

SmartBear – http://smartbear.com/

SDK Generators

APIMatic – http://www.apimatic.io

REST United – http://www.restunited.com

More Great Reads

A Practical Approach to API Design (2014)
D. Keith Casey Jr. and James Higginbotham
https://leanpub.com/restful-api-design

Restful API Design (2013)
Leonard Richardson and Mike Amundsen
Published by O'Reilly

API News/ Blogs

MuleSoft – http://blogs.mulesoft.org

MikeStowe.com – http://www.mikestowe.com

ProgrammableWeb – http://www.programmableweb.com

NordicAPIs – http://nordicapis.com/blog/

APIUX – http://apiux.com/

API Evangelist – http://apievangelist.com/blog/

API Handyman – http://apihandyman.io/

API2Cart – https://www.api2cart.com/blog/

API Workshop (VLOG) - http://bit.ly/APIWorkshop

Mike Amundsen – http://amundsen.com/blog/

Roy Fielding, Untangled – http://roy.gbiv.com/untangled/

Just for Fun

{"apis":"the joy"} - http://apijoy.tumblr.com

For a list of up-to-date resources, visit http://bit.ly/apiResources

Appendix: Is Your API Ready for Developers?

- [ ] You've taken the time to talk to potential developers and make sure your API meets their needs (see Chapter 2)
- [ ] You've defined your API in a spec like RAML (raml.org) and had developers test out your API to make sure it functions as expected (see Chapters 3, 4, & 5)
- [ ] You've reviewed your API to make sure it's consistent with best practices including resource naming, method design, content-type handling, etc. (see Chapters 7, 8, & 9)
- [ ] You have a way to manage API users, keys, throttling, provisioning and scaling your API (see Chapter 11)
- [ ] You've created an API Portal with extensive documentation or interactive documentation to help your developers use your API. Remember, by defining your API in RAML you can automagically generate your documentation and provide more interactive tools. (see Chapter 12)
- [ ] You've set up API Notebooks (apinotebook.com) to show your developers sample use cases and flows using your API (see Chapter 12)
- [ ] You've provided code libraries or links to services such as APImatic.io to help your developers get started without having to write custom methods to access your API (see Chapter 12)
- [ ] You've set up a community of support to help your clients use your API and get their questions answered by technical experts (see Chapter 12)

About the Author



Michael Stowe is a professional, Zend Certified Engineer with over 10 years experience building applications for law enforcement, the medical field, nonprofits, and numerous industrial companies. Over the last two years he has been focused on API design and ways to improve industry standards, speaking at API World & API Strategy and Design, while being a leading voice for RAML and Spec Driven Development.

@mikegstowe
http://www.mikestowe.com

About MuleSoft

MuleSoft provides the most widely used integration platform for connecting any application, data source or API, whether in the cloud or on-premises. With Anypoint Platform™, MuleSoft delivers a complete integration experience built on proven open source technology, eliminating the pain and cost of point-to-point integration. Anypoint Platform includes CloudHub™ iPaaS, Mule ESB™, and a unified solution for API management, design and publishing.

@mulesoft
http://www.mulesoft.com

Design Great APIs

APIs are fundamentally changing the way that companies are doing business, creating unprecedented opportunities for innovation and providing new ways to engage with your customers, partners, and employees.

Undisturbed REST digs into the questions that one must answer to be successful in building and designing their API, while also covering modern design techniques and explaining more complex concepts such as REST and hypermedia as the engine of application state (HATEOAS).

By following the carefully laid out approaches and taking advantage of the best practices shared in this book, you'll be armed with the knowledge to build an API your customers will love, as well as one that can be easily integrated and extended.

“ After two years of sharing how to build APIs, I realized the challenge isn't in building an API - there are tons of frameworks and tools to help you with this - but rather in designing an API that can meet all of your needs while being long-lived. Undisturbed REST is the culmination of many of my own talks, experiences, challenges, and the lessons I've learned to not only make the process efficient, but to create an API that is flexible enough and well thought-out enough to stand the test of time. ”

- Michael Stowe, Author

MuleSoft

PUBLISHED BY MULESOFT INC.
WWW.MULESOFT.COM