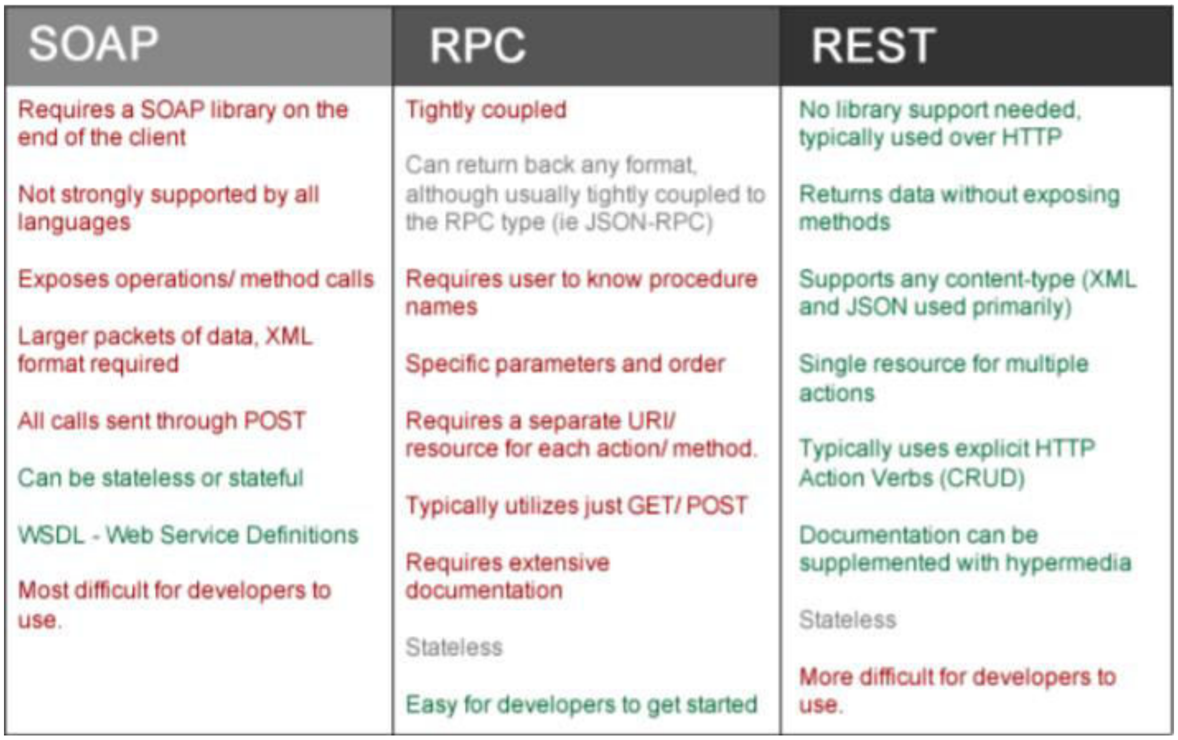
What is a REST API?

REST or RESTful API design (Representational State Transfer) is designed to take advantage of existing protocols. While REST can be used over nearly any protocol, it usually 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 design. REST API Design was defined by Dr. Roy Fielding in his 2000 doctorate dissertation. It is notable for its 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.

This freedom and flexibility inherent in REST API design allow you to 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.

However, there are drawbacks to REST API design. You can 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 can hinder your efforts without even realizing it.



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	

Understanding REST API Design

While most APIs claim to be RESTful, they fall short of the requirements and constraints asserted by Dr. Fielding. There are six key constraints to REST API design to be aware of when deciding whether this is the right API type for your project.

Client-Server

The client-server constraint works on the concept that the client and the server should be separate from each other and allowed to evolve individually and independently. 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.

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. Identifying information is not being stored on the server when making calls. Instead, each call has the necessary data in itself, 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.

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. 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, models, or 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.

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 API design, 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.

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. Additionally, the layered system 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. 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.)

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.

For more information about REST API Design, check out the eBook Undisturbed REST: A Guide to Designing the Perfect API.

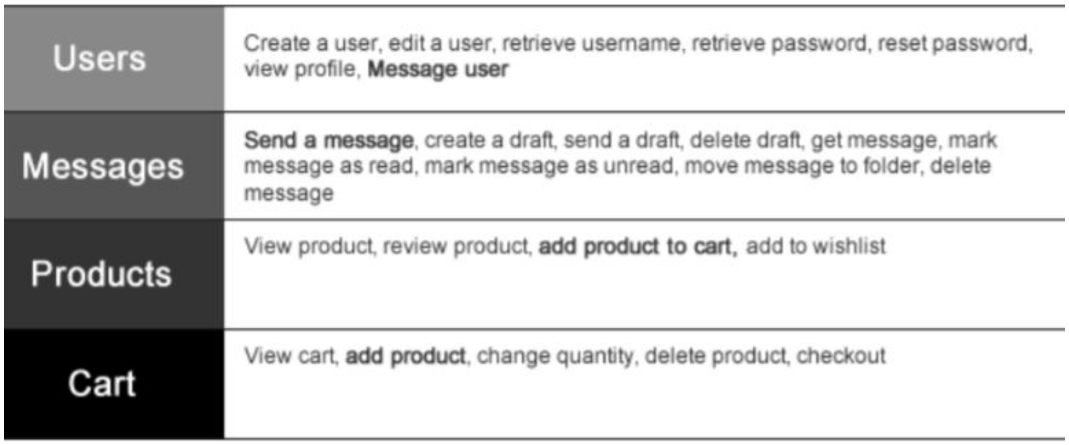
Planning Your API

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 woulda 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 a actually 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.