Amicable APIs

am·i·ca·ble/'amikəbəl/

Adjective: Having a spirit of friendliness; without serious disagreement or rancor.

Introduction

The origin of this paper stems from the confusion I felt when I came to realize that SOAP wasn't really as wondrous as I had been led to believe after reading numerous books on the subject - from simple SOAP definitions to advanced tomes by Thomas Erl.

A few years ago I had quickly written off the concept of Representational state transfer as silly academia just trying to formalize the simple concept of browsing webpages. Also after adding an http GET interface for address lookup to the .Net system I was then in charge of, my thoughts of REST was that of "been there done that". Except I had not been there at all, not even in the ballpark. All I had done was to utilize a crudely implemented XML-RPC interface (which unfortunately even today is all too common). Back then I still thought that the automagically generated client proxy classes that my VisualStudio.Net gave me after pointing it to the appropriate WSDL was the top of the API evolution.

What finally made me see the light and allowed me to formulate the thoughts described herein was the frustration felt after trying to have several internal and external teams use a fairly complex SOAP interface that I took part in creating. It was decorated with all sorts of WS-* protocols to put the local network security powers to be at peace, including WS-Security, SAML tokens, a Secure Token Service and all sorts of network and application firewalls. At the same time I was trying to create a public internet facing REST version of our SOAP interfaces to align with some other existing REST APIs already in place. To better understand what I was doing I got the O'Reilly book "RESTful Web Services" by L. Richardson and S. Ruby. As I was reading this book something "clicked" inside and I came to understand at a fundamental level what was wrong with SOAP and why REST (and ROA) was a far more superior paradigm. Some concepts seem to resonate with us at a deeper level rather than just being another set of rules to follow. This has only happened a few times in my career. Fundamentally understanding why Agile methodologies are better than Waterfall is another such Eureka moment.
If you haven't read "RESTful Web Services" yet, please do. It's the only text on REST that you really need.

Origins of Interfaces

It's hard to say where and when the first Application Programing Interface was introduced *(I really tried to Google it)*, but one key observation regarding interfaces in general was made as early as 1968 by Melvin Conway and is commonly referred to as Conway's law: *"Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure"*. A more clear way to rephrase this could be: *Anywhere there's a team boundary, you will find a software boundary*. More importantly is that from Conway's law follows that *the quality of a system interface typically reflects the quality of the communication between the groups involved.*

Conway's law is not a mere observation, it is a profound truth about group dynamics and my experience is that it is equally valid in scenarios involving either internal or external teams. So it's important to realize that system interfaces are not selected arbitrarily, but rather reflects the properties of the group in charge. Are they open to ideas from outsiders? Are they protective or secretive? How do they relate to other teams, both inside the organization but also outside of it, etc... The answer to those type of questions will have a bigger impact on the technical properties of the system interface than the actual technology stack used to create the system itself.

Modes of communication

In the light of Conway's law, it is interesting to look at what constitutes high quality communication between groups. In system development methodologies one often talks about the concepts of ceremony and precision. Traditional waterfall methodologies as practices by major consulting firms, contains a lot of processes, procedures and documentation that can be plagued by both high ceremony and high precision. More contemporary methods that typically are labeled "Agile" seems less so. Agile methodologies tries to emphasize real honest communication between constituents, where understanding rather than prestige seems to be the goal. Similarly, this should be the goal in any exchange between teams tasked with making their systems interface.

Alas, be aware that natural langue can be somewhat treacherous, just look at the seemingly innocent sentence: *"Mary had a little lamb"*. Think about how this sentence changes meaning depending on which word is emphasized when spoken. *"**Mary** had a little lamb"* (what about Ben or Lisa?). *"Mary **had** a little lamb"* (she no longer has one?). *"Mary had **a** little lamb"* (but not two or three?). *"Mary had a **little** lamb"* (so it was not medium sized?). "Mary had a little **lamb**" (but no cows or dogs?) or "Mary had a little lamb**?**" v.s. "Mary had a little lamb**!**" This somewhat naive exercise tries to show the futility in trusting written requirements specifications. Handing someone a requirement specification is probably the worst way one can go about building a system. It gives the author a false sense of accomplishment at best and is something to hide behind at worst (what do you mean you don't understand? it's all in the spec!)

This is not about requirements analysis, but since I brought it up - the real goal of requirement analysis is to re-create the domain model that exists in the heads of the domain experts into the collective consciousness of the development team (sometimes it requires a good amount of “interviewing” before the concept of a formal model will emerge). This process is best done as a team effort, face-to-face using props like whiteboards, markers, post-its, etc...

So if formal meeting protocols and meticulously crafted and physically signed agreements is not the answer to high quality inter team communication, what is?

The answer might be simpler than you think. In the early days of my career I was fortunate enough to come across a piece of hardware that went under the name LISP machine. One of the wonders of the LISP machine was it’s LISP interpreter which was nicknamed **DWIM** - *Do What I Mean*. It parsed the input and tried to figure out what you really wanted to type, despite misspellings and miss-matching parenthesis pairs.
It so happens that this concept also exists in inter human communication - it’s called amicability...

I’ve been in countless meetings involving multiple teams where it was very obvious to me what the current speaker was trying to communicate but was astonishingly met with a stare of non-understanding or even worse, rude passive-aggressive remarks. Most likely because the recipient didn’t think it was in his or hers interest to actually gain understanding, their agenda was that of obfuscation and deceit or afflicted by the all to common *“Not Invented Here”* syndrome.

Truly great interfaces can only be created and consumed when participants communicate with an open mind and a willingness to understand each other. One helpful advice I’ve taken to heart from the book *“7 habits of highly effective people”* by Stephen Covey is *“Seek first to understand, then to be understood”*.

Amicability applied to interfaces

The same way the mode of human to human communication can either be ceremonious and/or obfuscated v.s. open and amicable, so can the style of a system interface be. On the one hand it can be full of precision, ceremony and intolerance (think SOAP interface using WS-*, WDSLs and XML schema validation) v.s. robust and forgiving (think HTTP using only simple verbs supporting multiple data representation formats).

But using XML-Schema validation and strong run-time typing of business objects is a good thing... or is it? And besides it’s all controlled and interpreted by our modern Eclipse and VisualStudio.Net IDEs anyway?
Lets take a look at what typically happens when a system needs to consume a new API to connect to another system. Consuming an API really consists of two phases: 1) establish system connectivity and get something back (SSL handshake error, SOAP exception, 404, actual SOAP envelope, etc...). Call this syntactic connectivity.

Then 2) establish information exchange (actually passing and consuming data back and forth). Call this semantic connectivity.

Syntactic connectivity is typically done in conjunction with some network and security engineers depending on the size of the moat and fortress your current organization has established. This can take anything from a couple of hours up to a month, depending on the complexity of network security, the level of inter team amicability, difference in location/time zones and to some extent technical competence. And as the system build propagates through deployment environments (sandbox, development, QA, UAT, production-staging, production, etc...) this typically needs to be done all over again since the security postures of these environment tends to be very different.

One of the biggest issues involving syntactic connectivity once you move outside the comfort zone of basic SOAP over http:80 is interoperability. Most WS-* protocols are so full of precision and complexity that organizations tend to get 3rd party software and/or appliances to deal with it. Anyone ever involved in trying to connect two organizations using different vendors to deal with SAML tokens knows what I mean.

Now over to semantic connectivity. Since no one is really manually designing their WSDL definitions first, most SOAP solutions tend to reek of framework and implementation details that stems from automatically created WDSL definitions. Some of the constructs used by Spring or .Net might be completely foreign to the consuming system and it's developers. Same thing with your internal business objects or even worse, database model. It almost requires a fortune teller reading tea-leaves to understand the internal structure of an old school relational database using cryptic table and field names based on some archaic data dictionary that no one dares to touch ever since the last data architect left 5 years ago. Not to mention the reasons for the current normalization effort (or lack thereof), Boyce-Codd normal form anyone? *(BCNF is a.k.a 3.5 normal form)*. It's an illusion that system interfaces are consumed by computers. APIs are first and foremost consumed and interpreted by humans trying to build support for the API into their own systems, only after that are APIs consumed by computers.

Well, it is what it is, right? Not so much when your competitors who were smart enough to use a system interface based on principles of amicability spent all of a couple of hours to establish syntactic and semantic connectivity. Compared to the time it can take when dealing with unforgiving SOAP interfaces it can spell missed opportunities and business disaster.

Is REST Amicable?

So why is REST amicable and SOAP not?
REST isn't necessarily more amicable than SOAP, but it's a better candidate for building amicable interfaces than SOAP for the following reasons:

- The REST metaphor aligns with the use of web browsers
  Web browsers have been around for 15 years and it's use is ubiquitous. No software installation needed. Just type the URL and hit return. Done!

- Developers can interact with a REST interface without documentation
  The amount of documentation needed to explain how to call a REST based API based on a given URI is zero. Everyone knows how to use a web browser. In fact most websites can be viewed as REST based APIs returning the MIME type of text/html. Also a lot of information can be digested by just looking at what is returned.

- Since REST is limited and standardized, it's easy to experiment with
  The http protocol is purposefully limited and standardized. Most constraints (e.g. http verbs, return codes, header variables, etc...) are known in advance (or can easily be Googled) so developers are left with filling in the remaining blanks by playing around with different types of URIs, resources and parameters. By looking at resulting data and further digging in by following resource links they can gather a lot of understanding in a short time.

- REST datasets doesn't have to conform to a predefined schema
  No support for XML-schema validation allows for robust interfaces that still functions as usual even after new data points have been added. Besides, who uses XML anymore, haven't you heard of JSON?

Let me develop my points about minimal documentation and no schema validation a little further.

Of course there need to be some level of documentation, especially to explain the part of the API that is using POST, PUT and any type of security features (e.g. basic or digest authentication, OAuth 1.0a, OAuth 2, etc...). However humans learns best from examples. Our brains are basically highly specialized pattern matching machines that loves to validate (or invalidate) an initial set of assumptions with actual examples in order to establish a more refined abstraction model. Being fed this abstract model directly typically doesn't work as well.

Consider how data returned by an API is typically used in any given system. It either is passed around internally through different levels of procedures/abstraction layers or is being displayed on a screen. API data is less likely to be stored internally since it typically is mastered by the external system that initially was called to access it.

As long as it is being passed around the system, the data schema doesn't really matter (it's just a pointer to a block of memory on the system stack or heap). Similarly, when displayed on a screen all that matters is that it can be typecast/marshaled to a character string (or whatever string class your flavor of framework is using). Typically only a subset of all the API data is being displayed. The conclusion of this is that datatypes doesn't really matter, rather simpler is better. Most modern programming environments in use today that interacts with REST based APIs can with only a couple of lines of code convert any JSON or XML structure to a native data structure consisting of only object arrays, object dictionaries and strings (although most also handles integers, floats and Booleans as well). Data is then typically retrieved from a dictionary by providing the key value that corresponds to a specific data string. Note that these structures although simple by themselves, can create quite complex data structures with multiple levels of dictionaries containing other dictionaries or arrays, etc. Also note how adding new data nodes to this structure will have absolutely no effect on the robustness of the existing use of it. I.e. the API is invariant as far as adding new data to it. This in combination with the fact that only a few datatypes are involved is one of the key aspects of REST being a great candidate for an amicable API.

Increased Amicability

How then would one go about and design an amicable API based on the foundations of REST and ROA? The following is a list of amicable design patterns in no particular order.

- Design the API with ROA in mind (Resource Oriented Architecture) as defined in the book "RESTful Web Services". Don't just settle for using random URIs and some XML like data. If you design the API with ROA in mind, others familiar with this style will be able to pick up on your API in no time.

- Follow the KISS principle as much as possible. Keep URIs as simple as possible. E.g. http://SomeDomain.com/PluralFormOfResource/IdentifyingElement.DocumentType. Only design in terms of resource names and CRUD operations. Never use URIs that contain verbs, that will take you to XML-RPC land faster than you can say REST. Again, it's really hard for someone to guess your verb URIs v.s. knowing that the only resource types supported are X, Y and Z. Don't be afraid to use sub-resources, they are very helpful in allowing more precise operations. E.g. to update order quantity for a line item on an order, it would be easier to access the line item through a URI and only provide the updated data for that (i.e. ../orders/12345/items/2) rather then PUTing the whole changed order to the URI ../orders/12345. The reason for using the plural form of a resource name in the URI is amicability but also esthetics (it makes sense to use the form ../orders.json to retrieve all orders and by consistently using plural form, there will be less guessing).

- No surprises, try to adhere to the official usage of HTTP result codes and variables (and use as few of them as possible). But please provide some meaningful error descriptions when returning any 400 or 500 codes.

- Allow for alternative URIs to access the same resource. This helps with the amicability since it's more likely a user would hit on a valid URI when playing around with the API. E.g. accessing a user could be done with the canonical (preferred) URI http://somedomain.com/users/*userid*.xml or equal valid URIs http://somedomain.com/users/*username*.xml or http://somedomain.com/users/emailaddress.xml or http://somedomain.com/users/cellphonenumber.xml etc... Sometimes it's entirely possible to figure out what the identifier is (i.e. id, name, email or phone#) by applying some clever regular expression matching on the server side. E.g. ids are [0-9]+ ; names are [A-Z][a-z]* ; emails are [0-9|a-z]+'@'[0-9|a-z]+'.'[a-z] + ; phone numbers are [0-9][0-9][0-9]'-'[0-9][0-9][0-9]'-'[0-9][0-9][0-9][0-9]. Formats will vary slightly between platforms.

- Allow for multiple document (data) formats (at least XML, JSON and HTML) and multiple ways to request it (e.g. ending the URI with the document type .xml, .json, etc or by providing the Accepts request header). If you allow the calling client to request which data format to receive, it makes the API more versatile. Consider supporting a simple HTML table tag version for each call to allow software implementers to simulate full end-to-end functionality. To easier support PUT, POST and DELETE operations, you can add some scaffolding forms whose POST operations with accompanying *application/x-www-form-urlencoded* data pairings can then be appropriately interpreted by the API backend. If the API is an extension of an existing website, the client developers should be able to perform most website features using the API in html mode together with the scaffolding forms. Some additional formats to consider could be text/plain, text/cvs and why not .plist (xml format that allows native NSArrays and NSDictionaries in iOS apps to be populated directly from a http response using only one line of code).

- Minimize the need for versioning (there is no need for new versions as long as all changes are backwards compatible, i.e. additions only).

- Design with worst case latency in mind. Think about how API data access patterns could/should be different when a consuming client app is connected via a mobile edge cellular network (i.e. even worse than 3G). v.s. sitting at Starbucks using a local WiFi connection.

- Strictly follow the intention of POST vs PUT. Again, not to be a ROA fanatic, rather to support the case for predictability. POST is done to create/update info when a unique URI can't be known (e.g. creating an order that will return a newly assigned order number. PUT is used when creating/updating data using a canonical URI that uniquely identifies a resources is known (e.g. creating a new tax payer record using the SSN as the key - I know, it's a pretty bad example in these times of privacy concerns).

- Provide a stable test environment. It is important to understand the difference between a QA environment for the API itself and a QA environment for client connectivity. QA for the API itself is typically a pretty volatile environment with new deployments up to several times a week. This should be utilized to perform quality assurance on new API features. When clients are in dev/test, they need to be able to connect to an environment that looks and feels like the current production environment for the API, both in terms of security posture, features and actual data returned. Consider providing a refreshable data storage that can be reset to a known set of test data fixtures when needed. The test environment could be the same as production with an additional URL parameter or request header to inform the API that this is a test call v.s. actual production. The advantage with a separate environment is that it can also support “preview” functionality in advance of features being deployed to production.

- If things do fail, provide meaningful error message that will identify the exact layer/component/subsystem where the error occurred. This might not mean much to the client side team, but it sure will help when they contact you because the clients keep crashing. If all they can tell you is that they are getting the response “500 - Internal Server Error” no one will be happy.

Authentication and Security

Okay, there’s a reason why this topic has been avoided so far. Authentication and security standards are pretty lame and confusing as far as REST is concerned. I agree that SOAP has more detailed defined standards for this, but said standards are extremely complicated and tedious to implement, both on the server side and client side. That being said, here are my recommendations for amicable authentication and security practices when building a REST API:

- First of all think about the reasons you might have for adding authentication and security features. Is the data involved sensitive from a corporate financial point of view (e.g company secrets, credit card info, etc..), is it of privacy concerns (e.g. emails, SSNs, etc...) or is it more a case of being able to control and/or throttle API resource usage or all of the above? These questions will help you better understand why a certain authentication or security feature might be helpful or just a hindrance.

- The most basic concern with any REST API is that it should be safe and idempotent. Safe means that no matter how many times you submit an HTTP GET request, there should be no data updates at all. GET is strictly considered a read operation. If your GET is changing data you are doing XML-RPC not REST, period. Idempotent means that no matter how many times you submit an HTTP GET, HEAD, PUT or DELETE the effect should be exactly the same as when you submit it only once. For POSTs, there’s no guarantee of idempotence, so the more you can encourage the use of your PUT operations v.s. POSTs the better.

- There's really no reason for not using SSL (HTTPS) other than in extreme latency or weak client processor scenarios.

- Even when using SSL, certain operations are so sensitive that you don't want to risk that someone could intercept them, change them and then resubmit the changes. In these cases try to use a cryptographic hash. One great example is using HMAC-SHA1 which uses a shared secret between the server and client. Any call has to contain a calculated hash that depends on the complete message (URI and body) so that it will be invalid when checked on the server side, should any manipulation have occurred. Also, the HMAC-SHA1 algorithm is being used by the fairly common OAuth 1.0a protocol so most environments should have plenty of code libraries available to implement the cryptographic hash part of OAuth 1.0a (forget about the rest of it, it's way too complicated).

- If you require an API key to be provided with each call, make sure your reasons for doing so are clear and meaningful and not something you do just because. Reasons might be tracking/billing individual accounts, apps, companies, etc... or allow for fine grained throttling of API traffic.

- Consider using the appropriate HTTP request/response headers (i.e. *WWW-Authenticate* and *Authorization*) for any authentication/security features rather than lazily just tack them on as URL parameters (not good since they will be part of any web browsers history audit trail).

- If using SSL is too taxing, consider obfuscating sensitive data elements using Base64 encoding (or similar) to prevent any outsider from peeking. Base64 can also be good when there's a risk that the data itself will interfere with the document structure (e.g. using the '<' or '>' characters between XML element tags). Similarly you could choose to only apply a HMAC-SHA1 hash to certain data elements only v.s. the whole HTTP request.

- Never ever submit userids and/or passwords in clear text, at least not as a URL parameter and especially when NOT using SSL. Even submitting these in the HTTP body or request header should be a big no-no. For simple cases there's the HTTP standard called Basic Authentication or better yet, use the one called Digest Authentication. Great from an amicability point of view since they are fairly common and well documented.

- For 3rd party access to a user's data, there's OAuth 2 (used by Facebook) and OAuth 1.0a (used by everyone else). OAuth is well documented and common but unfortunately, at least OAuth 1.0a is very tedious and complex to learn. OAuth 2 is less so and should be preferred over OAuth 1.0a from an amicability point of view.

- If you organization already are using some other token or claims based standard for user authentication and/or security, don't be afraid to come up with your own protocol format to use with *WWW-Authenticate* and *Authorization*. Once you are comfortable

with using the HTTP standard to implement your API, you'll see that it's not that hard to come up with your own standards for authentication. Just try to make it logical and amicable if at all possible (at least provide lots of documentation and test scaffolding).

- Consider providing a "naked" implementation of your API for testing purposes to allow for easy learnings using a web browser. Just make sure to strip out any secret or sensitive data.

Client-side Amicability

So far we have only been focusing on the server side, what about clients, what can they do to increase the probability of success? Here's a few pointers:

- Make sure that client side code can recover from network interruptions. It can be as simple as enclosing any API interaction in a *"try/catch"* exception handling block.

- Don't use any variables requiring strong compile time and/or run-time type checking. Assume that API data can be augmented at any time. Use object arrays and dictionaries to manage returned data. Allow objects to be strings, integers, floats or booleans (or treat all data elements as strings).

- Don't hardcode any initial URIs or security/authentication parameters. Use boot-strapping where all initial values are provided by a service fully under the client side team's control. Provide an interface where these can be instantly updated without any need for redeployment.

- Consider implementing client side caching to provide an uninterrupted experience should a network issue arise.

- Create a lot of automatic tests for scenarios that involves the API. Run these tests frequently to make sure that the API behaves as expected.

Final thoughts

Be aware of the cruelty of the 2nd law of thermodynamics. It is as fundamental as any other natural power. It affects software systems as much as any other natural phenomena out there. In it's most basic form it predicts with a 100% certainty that any system left unattended will sooner or later fail. Adhering to the principles of amicability will for sure prolong such a fate. Another important safety measure is to try to minimize the total number of separate systems involved in any one call-chain interaction between the client and the backend system that the API is supporting. Let's look a little closer at the math behind the above claims.

The best analogy for understanding the effects of the 2nd law of thermodynamics on any computer system is statistical mechanics. According to this theory the number of possible combinations of values that a system's internal state (instance variables, associated database records, etc…) can have can be defined as the possible micro states of that system. These micro states can then be grouped into clusters so that each cluster defines a high-level state (e.g. active, inactive, working, crashed, waiting, etc…). Each such cluster can then be defined as the possible macro states for that system. This micro/macro state definition corresponds directly to the entropy concept defined by statistical mechanics. The entropy of a macro state is defined as the number of micro states that it includes. In combination with the 2nd law of thermodynamics we get: Unless work is done, a system's entropy will always increase until it gets to a macro state with the highest possible entropy. I.e. if a software systems is not maintained each of the components will in time always reach the most probable macro state (which typically is "crashed").

The reason for minimizing the number of involved systems in any call chain is equally simple. Say that we have 6 systems involved (e.g. client, network infrastructure, application firewall, API server, business object server, database server) and each system have a probability of "working just fine" equal to 90% (or 0.90). Then the probability for the total call chain to "work just fine" is equal to: 0.90 x 0.90 x 0.90 x 0.90 x 0.90 x 0.90 = 0.53 = **53%**