Application Development

REST vs RPC: What problems are you trying to solve with your APIs?

October 15, 2018

Martin Nally
Software Developer and API designer,
Apigee

A fairy ring is a naturally occurring circle of mushrooms that grows in forested areas or grassland. In folklore, fairy rings have magical properties and superstitious people carefully avoid disturbing them. There's an old joke about the farmer who was asked why he went to such lengths to avoid ploughing up fairy rings. He replied, "because I'd be a fool if I didn't."

Many people would say the same thing about why they build APIs. In fact, it is important to think about the fundamental problem you are trying to solve with your API because the style of API you create and the technologies you choose should depend on your answer.

Procedures, also called functions, have been the dominant construct for organizing computer code ever since FORTRAN II introduced the concept in 1958. All mainstream modern programming languages that are used to produce and consume APIs—for example Java, Javascript, Python, Golang, C/C++ and PHP—use procedures as their central organizing construct[1], so it is not surprising that procedures have also been the dominant model for designing and implementing distributed APIs for decades, in the form of Remote Procedure Calls (RPC).

If you ask most software developers why they define and build APIs, they are likely to explain that they have an application that is implemented as multiple distributed components, and those components call each other's APIs for the complete application to function. They may also say that they are implementing the API of a service that is used by multiple applications.

When developers design APIs to solve these kinds of problems, the solution characteristics they will typically prioritize are ease of programming for both the client and the server, and efficiency of execution. RPC is a good match for these priorities. RPC fits very well with the thought processes and skills of programmers on both the producer and consumer side of an API. Calling a remote procedure is usually syntactically the same as calling a normal programming language procedure,

implementations also tend to be efficient—the data that is passed between the client and the server is usually encoded in binary formats, and the RPC style encourages relatively small messages (although some care has to be taken to avoid overly chatty interactions).

If RPC is such a good fit with the rest of software development, why is there another popular model for APIs—REST—and why is there so much controversy about which to use?

Unsolved problems

Communicating between two components in a distributed system is largely a solved problem—there are many successful technologies available for different levels, like TCP for basic data transfer, RPC or HTTP as programming models, and various algorithms that deal with consistency of state in distributed systems. That doesn't mean it is easy to develop distributed systems, but the problems and their solutions are reasonably well-known.

By contrast, there are unsolved problems in software that have huge economic impact. Two of these are the fact that almost all software is extremely difficult to change, and that software systems are difficult to integrate. Both problems are relevant to the discussion of APIs because they help motivate an alternative model to RPC.

change

Unless you are working in a startup that is only a few months old, it is almost certain that one of your organization's most significant problems is dealing with a legacy of software that no longer fits its current needs or directions. In almost all cases, this software is too valuable to abandon, and too difficult to change. It is likely that a very large part of the organization's development budget is consumed by this reality.

One of the primary reasons that software is difficult to change is that basic assumptions are propagated through code from procedure to procedure. Some of those assumptions are technical assumptions, like what storage technologies are being used or what error or failure conditions are possible, while other assumptions concern the basic use-cases of the application, like which concepts are used with which others in what ways.

It isn't easy to characterize exactly why software is so brittle, but unless you're new to software development, you've almost certainly found yourself in the situation where relatively straightforward-seeming technical or functional changes prove to be extremely difficult to make, because assumptions in the code are broadly distributed and hence difficult to change.

efforts to improve software flexibility through better modularity without seeing much fundamental improvement—software remains stubbornly rigid and difficult to change. One of the lessons we can take from this is that technical and business assumptions pass through procedure boundaries like water through a sieve. This is largely true of remote procedures as well as local procedures.

Integration is a major opportunity and problem

When you first start writing software to automate a particular problem domain, the majority of the software you write will focus on automating basic features and workflows. As the software matures, more value comes from integrating and augmenting existing systems, rather than building brand new ones. For example, many companies have development initiatives to create a more integrated experience for customers interacting with their different systems, or to help the business see an overall picture of its customers, suppliers, or partners across multiple systems.

When businesses open up APIs to their systems, they sometimes have a purely tactical goal, like enabling the development of a mobile application. More visionary businesses, however, open up APIs

other words, the primary motivation for many businesses in creating APIs is to enable third parties to create integration applications of which their systems can be a part.

For example, both travel and expense reporting applications are major feats of integration bringing together reservations of multiple services (transport, lodging, food, entertainment), approvals, financial records and so on, all across different companies. Anyone who has worked on an integration project knows that they are hard. One difficulty is the sheer variability of the interfaces and technologies that have to be integrated. Another is that many of the systems don't even have quality APIs.

REST APIs, and how they can help

The global success of the world-wide web has led to a lot of interest in an alternative model for APIs—REST.

REST itself is a description of the design principles that underpin HTTP and the world-wide web. But because HTTP is the only commercially important REST API, we can mostly avoid discussing REST and just focus on HTTP. This substitution is useful because there is a lot of confusion and variability in

agreement on what HTTP itself is. The HTTP model is the perfect inverse of the RPC model—in the RPC model, the addressable units are procedures, and the entities of the problem domain are hidden behind the procedures. In the HTTP model, the addressable units are the entities themselves and the behaviors of the system are hidden behind the entities as side-effects of creating, updating, or deleting them.

One of the remarkable characteristics of the world-wide web is that every address on the web exposes exactly the same API—HTTP. (In the REST description of HTTP, this is called the "uniform interface constraint.") This means that to navigate the entire world-wide web, you only need to know a single API—this fact is what made possible the development of the web browser. Unfortunately, many of the APIs that claim to be RESTful layer a lot of proprietary concepts on top of HTTP. Essentially, they invent their own API and use HTTP as a lower-level transport layer, rather than using HTTP directly as it was designed. In fact, there's so much variability in the way that people use the term REST in the context of APIs that it's difficult to know what they mean by it unless you know them well.

Because HTTP is already so widely known, there's a lot less to learn about an API that uses HTTP directly than an RPC one. Learning an RPC API is very similar to learning a programming library. The interface of a programing library is typically made up of many procedure signatures that each have to be learned,

not even within the same library).

By contrast, learning an API that uses HTTP directly is like learning a database schema. Every database managed by the same database management system, whether it's Postgres, MySQL, or Spanner, has exactly the same API, so you only have to learn that API once. For an individual database, you only have to learn the tables and their columns[2], and their meanings; compared to a typical programming library, there is much less detail to learn in a database. An API that uses HTTP directly, like a database, is mostly defined by its data model. What about querying, you ask? It's true that complex queries—beyond simple create, retrieve, update and delete—are important in APIs as they are in databases, and that HTTP does not give us a standard query syntax for its API in the way that a database management system does, so there is typically more to learn that is specific to an HTTP API than to a database. But it's still much less than a corresponding RPC API, and the query syntax exposed by most APIs is much simpler than a database management system's. (One exception might be if the API includes something like GraphQL, in which case you have the compensating benefit that the query language is the same for many APIs.)

If an API uses HTTP simply and directly, it will only have to document three or four things. (And if an API requires you to read a lot of documentation to learn how to use it, then it is probably not using

1. A limited number of fixed, well-known URLs. These are analogous to the names of the tables in a database. For optional extra credit, make all the fixed URLs discoverable from a single one.
2. The information model of each of its resources, i.e., the properties of each type. This is analogous to the columns of a database table. Relationships between entities are expressed as URL-valued properties.
3. Some indication of the supported subset of HTTP, since few APIs implement every feature of the protocol.
4. Optionally, some sort of query syntax that enables efficient access to resource data without fetching whole resources one at a time. API designers are endlessly creative in how they allow queries to be encoded in URLs—my favorite option is to use only a query string appended to the well-known URLs defined in 1. above.

If the API is using HTTP properly, clients already know everything else they need to know because they already know HTTP.

How HTTP helps with integration

APIs just used HTTP simply and directly, because then the application only has to know HTTP, rather than a lot of different APIs. The data model exposed by each API may be different, but the mechanisms for accessing and changing the data will be the same for all of them.

An essential part of the problem in most integration applications is defining relationships between entities that are maintained in different systems. For example, flight reservations, hotel reservations, car reservations, credit card payments, and approvals all need to be linked together to manage a trip reservation or its reimbursement. In order to link these entities together, each must have a clear identity by which it can be referenced outside of the application in which it is housed. The world-wide web standards define the concept of a URL, an identifier for a resource that is valid everywhere (hence the name world-wide), not just locally in the API of a particular system. This is very helpful for integration applications because the problem is already solved for all entities that are accessed by APIs that use HTTP directly. By contrast, in RPC-based APIs the identity of an entity is almost always expressed in a form that is local to the application that houses it, putting the burden onto the developer to define an identity for each entity that is valid outside that application.

software easier to change

What about the difficulty of modifying software—can HTTP/REST help there too? We saw that the RPC model makes it very simple and direct for programmers to write a procedure in one program and call it from another. This is one of the characteristics that makes RPC so popular, but it also makes it easy for technology and use-case assumptions to flow easily from one application to the other, thereby coupling the two and making the system brittle.

HTTP/REST helps break that flow of assumptions, by forcing an intermediate translation from implementation procedures to an entity model. Procedures are no longer exposed directly at the interface to be called remotely; instead, API developers construct an entity model in between that disconnects the two sides. The entity-oriented model is not arbitrary; it is the conceptual data model of the problem domain as viewed by the client. It will likely have some relationship to an underlying storage data model, but is usually simpler and more abstract. How effectively the API entity model decouples the caller from the callee depends a lot on the skill of the model's designers, but the mere presence of the translation layer increases the chances of meaningful decoupling.

based on entities. When an API is based on remote procedures, it tends to grow organically as one procedure after another is added to handle specific needs. When an API is realized as an entity model, there is less tendency for unbridled organic growth, because entity models typically have a greater degree of coherence and overall structure. Evolving an API based on an entity model requires you to explicitly add a new type, property, or relationship to the model, which typically forces you to think about how the addition fits with the overall model. I have no doubt there are some well-governed RPC APIs where each procedure is one tile in a carefully-drawn mosaic, and no tile is ever added or changed without considering its impact on the whole picture. But in my experience this sort of coherence is more difficult to achieve for RPC APIs, and much less common.

No gain without pain

It's true that using entity-based APIs rather than procedures can introduce additional cost in the form of design and implementation complexity and processing overhead. Whether or not this cost is justified depends on your API goals. If efficiency is your first priority, RPC may be a better choice.

There is also a shortage of people who understand how to design good HTTP/REST APIs. Unfortunately, we see many examples of APIs that attempt to

the model consistently. Some common mistakes are:

1. Using "local identifiers" rather than URLs to encode references between entities. If an API requires a client to substitute a variable in a URI template to form the URL of a resource, it has already lost an important part of the value of HTTP's uniform interface. Constructing URLs that encode queries is the only common use for URI templates that is compatible with the idea of HTTP as a uniform interface.
2. Putting version identifiers in all URLs. URLs that include version identifiers are not bad in and of themselves, but they are useless for encoding references between entities. If all your URLs include version identifiers, you are probably using local identifiers instead of URLs to represent relationships, which is the first mistake.
3. Confusing identity with lookup. HTTP-based APIs depend on the fact that each entity is given an identity in the form of URI that is immutable and eternal. In many APIs it is also useful to be able to reference entities by their names and/or other mutable characteristics, like position in a hierarchy. It is important not to confuse an entity's own URI with alias URIs used to reference the same entity via a lookup on its name or other mutable characteristics.

story

HTTP does not mandate a particular data format, but JSON is by far the most popular. JSON was originally designed to be a good match for the way JavaScript represents data, but all mainstream programming languages support it. JSON is a relatively abstract format that is generally free of technology-specific constraints. JSON is also a text format, making it simple to understand and debug. These are good characteristics if ease of adoption and change are your primary goals, but less optimal if you're aiming for efficient communication between tightly-coupled components.

Programmers using statically-typed languages like Java, Golang or C++ will commonly push to constrain the ways in which JSON is used to fit with the ways in which their programming language prefers to process it. Whether or not you should accede to their wishes depends on your goals for the API.

Pick the API style that fits your goals

My message is not that HTTP/REST is better than RPC. If the goal of your API is to enable communication between two distributed components that you own and control, and

might be excellent choices for designing and implementing your API.

However, if your primary objective is to make your software more malleable by breaking it down into components that are better isolated from each others' assumptions, or if your purpose is to open up your systems for future integration by other teams, then you should focus your efforts on HTTP/JSON APIs—provided you learn to use HTTP as simply and directly as possible. That way, anyone who knows HTTP from the standards documents or a multitude of less formal tutorials will be able to use your API, with no more documentation than a description of the API's entity model and a little bit of query syntax. I think that's a worthy goal.

For more on API design, read the eBook, "Web API Design: The Missing Link."

1. Special-purpose languages like HTML, CSS or SQL that do not have functions or procedures or don't use them as a central concept are sometimes classified as "programming languages", but they are not used to implement APIs, so they are not relevant to this discussion.

2. If your database management system is a NoSQL database like MongoDB, CouchDB or Cassandra, then the vocabulary here is a bit different, but the idea is the same.

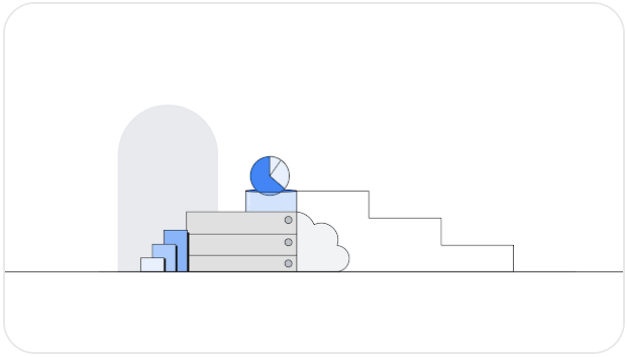
gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design

When designing a modern API, learn when to use RPC (gRPC), OpenAPI or REST.

By Martin Nally • 16-minute read

Posted in Application Development

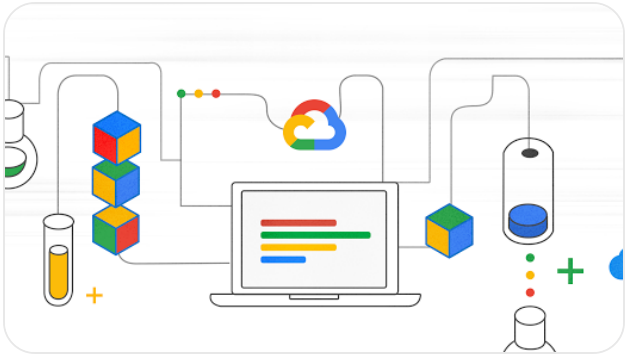
Related articles



Databases

Scaling Character.AI: How AlloyDB for PostgreSQL and Spanner met their growing needs

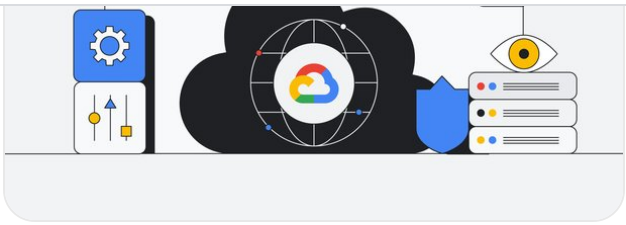
By James Groeneveld • 6-minute read



Application Development

No GPU? No problem. localllm lets you develop gen AI apps on local CPUs

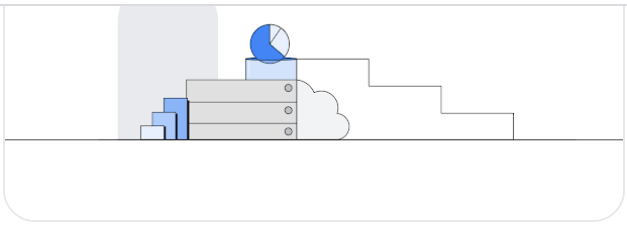
By Geoffrey Anderson • 6-minute read



Google Cloud

The overwhelmed person's guide to Google Cloud: week of February 1

By Richard Seroter • 3-minute read



Databases

Firestore Multiple Databases is now generally available

By Sichen Liu • 3-minute read