Understanding RPC Vs REST For HTTP APIs

September 20, 2016

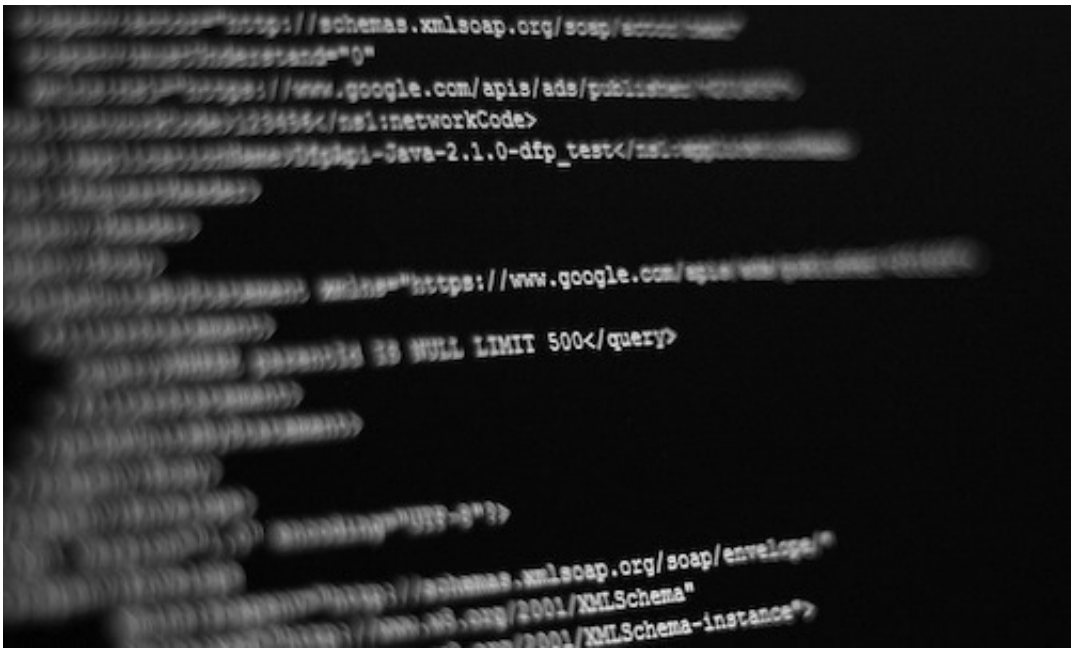
About The Author

Phil is a programming polyglot, author of , and a Software Engineer at WeWork. As a Brit in New York, he's always trying to …

Coding (589 articles)
Tools (167 articles)
JavaScript (164 articles)
Ruby on Rails (11 articles)

Table of Contents

For the last few years, whenever somebody wants to start building an HTTP API, they pretty much exclusively use REST as the go-to architectural style, over alternative approaches such as XML-RPC, SOAP and JSON-RPC. REST is made out by many to be ultimately superior to the other "RPC-based" approaches, which is a bit misleading because they are just different.

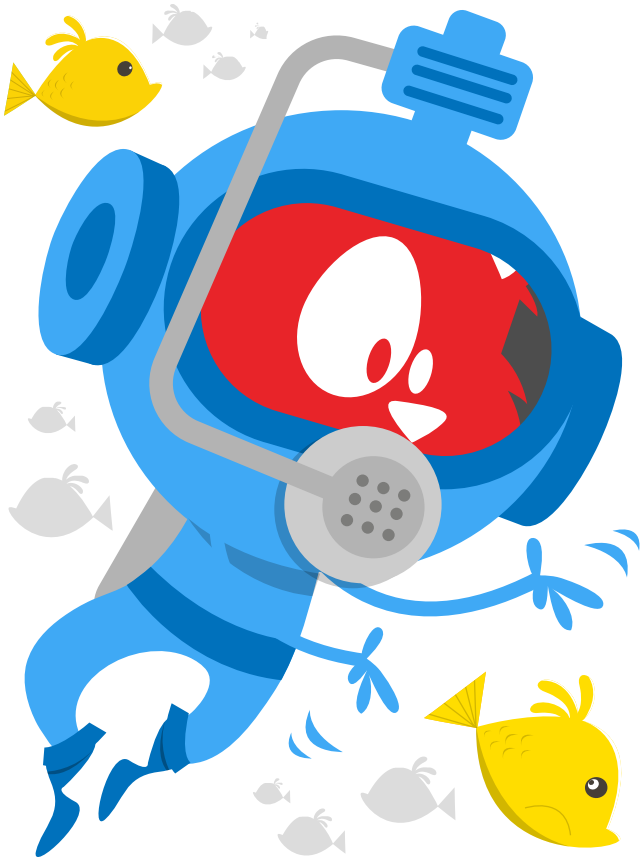


This article discusses these two approaches in the context of building HTTP APIs, because that is how they are most commonly used. RPC and REST can both be used via other transportation protocols, such as AMQP, but that is another topic entirely.

Further Reading on SmashingMag:

- HTTPS Everywhere With Nginx, Varnish And Apache
- A Beginner's Guide To jQuery-Based JSON API Clients
- How To Apply Transformations To Responsive Web Design

No need for silly advertising. At this point we just want to **thank you** for sticking around. This lil' website wouldn't exist without you. You are indeed quite… ahem, *smashing*. Happy reading! ;-)



REST stands for "**re**presentational **s**tate **t**ransfer," described by Roy Fielding in his dissertation. Sadly, that dissertation is not widely read, and so many people have their own idea of what REST is, leading to a lot of confusion and disagreement. REST is all about a client-server relationship, where server-side data are made available through representations of data in simple formats, often JSON and XML. These representations for resources, or collections of resources, which are then potentially modifiable, with actions and relationships being made discoverable via a method known as hypermedia. Hypermedia is fundamental to REST, and is essentially just the concept of providing links to other resources.

Beyond hypermedia there are a few other constraints, such as:

- REST must be stateless: not persisting sessions between requests.
- Responses should declare cacheablility: helps your API scale if clients respect the rules.

- REST focuses on uniformity: if you're using HTTP you should utilize HTTP features whenever possible, instead of inventing conventions.

These constraints (plus a few more) allow the REST architecture to help APIs last for decades, not just years.

Before REST became popular (after companies such as Twitter and Facebook labeled their APIs as REST), most APIs were built using an XML-RPC or SOAP. XML-RPC was problematic, because ensuring data types of XML payloads is tough. In XML, a lot of things are just strings, so you need to layer meta data on top in order to describe things such as which fields correspond to which data types. This became part of the basis for SOAP (Simple Object Access Protocol). XML-RPC and SOAP, along with custom homegrown solutions, dominated the API landscape for a long time and were all RPC-based HTTP APIs.

The "RPC" part stands for "remote procedure call," and it's essentially the same as calling a function in JavaScript, PHP, Python and so on, taking a method name and arguments. Seeing as XML is not everyone's cup of tea, an RPC API could use the JSON-RPC protocol, or you could roll a custom JSON-based API, as Slack has done with its Web API.

Take this example RPC call:

```
POST /sayHello HTTP/1.1
HOST: api.example.com
Content-Type: application/json

{"name": "Racey McRacerson"}
```

In JavaScript, we would do the same by defining a function, and later we'd call it elsewhere:

```
function sayHello(name) {

}


sayHello("Racey McRacerson");
```

The idea is the same. An API is built by defining public methods; then, the methods are called with arguments. RPC is just a bunch of functions, but in the context of an HTTP API, that entails putting the method in the URL and the arguments in the query string or body. SOAP can be incredibly verbose for accessing similar-but-different data, like reporting. If you search “SOAP example” on Google, you’ll find an example from Google that demonstrates a method named `getAdUnitsByStatement`, which looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <ns1:RequestHeader
         soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
         soapenv:mustUnderstand="0"
         xmlns:ns1="https://www.google.com/apis/ads/publisher/v201605">
      <ns1:networkCode>123456</ns1:networkCode>
      <ns1:applicationName>DfpApi-Java-2.1.0-dfp_test</ns1:applicationName>
    </ns1:RequestHeader>
  </soapenv:Header>
  <soapenv:Body>
    <getAdUnitsByStatement xmlns="https://www.google.com/apis/ads/publisher/v2
      <filterStatement>
        <query>WHERE parentId IS NULL LIMIT 500</query>
      </filterStatement>
    </getAdUnitsByStatement>
```

```
  </soapenv:Body>
</soapenv:Envelope>
```

This is a huge payload, all there simply to wrap this argument:

```
<query>WHERE parentId IS NULL LIMIT 500</query>
```

In JavaScript, that would look like this:

```
function getAdUnitsByStatement(filterStatement) {

};


getAdUnitsByStatement('WHERE parentId IS NULL LIMIT 500');
```

In a simpler JSON API, it might look more like this:

```
POST /getAdUnitsByStatement HTTP/1.1
HOST: api.example.com
Content-Type: application/json

{"filter": "WHERE parentId IS NULL LIMIT 500"}
```

Even though this payload is much easier, we still need to have different methods for `getAdUnitsByStatement` and `getAdUnitsBySomethingElse`. REST very quickly starts to look “better” when you look at examples like this, because it allows generic endpoints to be combined with query string items (for example, `GET /ads?statement={foo}` or `GET /ads?something={bar}`). You can combine query string items to get `GET /ads?statement={foo}&limit=500`, soon getting rid of that strange SQL-style syntax being sent as an argument.

So far, REST is looking superior, but only because these examples are using RPC for something that REST is more adept at handling. This article will not attempt to outline which is "better," but rather will help you make an informed decision about when one approach might be more appropriate.

What Are They For?

RPC-based APIs are great for actions (that is, procedures or commands).

REST-based APIs are great for modeling your domain (that is, resources or entities), making CRUD (create, read, update, delete) available for all of your data.

REST is not *only* CRUD, but things are done through mainly CRUD-based operations. REST will use HTTP methods such as `GET`, `POST`, `PUT`, `DELETE`, `OPTIONS` and, hopefully, `PATCH` to provide semantic meaning for the intention of the action being taken.

RPC, however, would not do that. Most use only `GET` and `POST`, with `GET` being used to fetch information and `POST` being used for everything else. It is common to see RPC APIs using something like `POST /deleteFoo`, with a body of `{ "id": 1 }`, instead of the REST approach, which would be `DELETE /foos/1`.

This is not an important difference; it's simply an implementation detail. The biggest difference in my opinion is in how actions are handled. In RPC, you just have `POST /doWhateverThingNow`, and that's rather clear. But with REST, using these CRUD-like operations can make you feel like REST is no good at handling anything other than CRUD.

Well, that is not entirely the case. Triggering actions can be done with either approach; but, in REST, that trigger can be thought of more like an

aftereffect. For example, if you want to "Send a message" to a user, RPC would be this:

```
POST /SendUserMessage HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"userId": 501, "message": "Hello!"}
```

But in REST, the same action would be this:

```
POST /users/501/messages HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"message": "Hello!"}
```

There's quite a conceptual difference here, even if they look rather similar:

- **RPC**. We are sending a message, and that might end up storing something in the database to keep a history, which might be another RPC call with possibly the same field names — who knows?
- **REST**. We are creating a message resource in the user's messages collection. We can see a history of these easily by doing a `GET` on the same URL, and the message will be sent in the background.

This "actions happen as an afterthought" can be used in REST to take care of a lot of things. Imagine a carpooling app that has "trips." Those trips need to have "start," "finish" and "cancel" actions, or else the user would never know when they started or finished.

In a REST API, you already have `GET /trips` and `POST /trips`, so a lot of people would try to use endpoints that look a bit like sub-resources for these

actions:

- `POST /trips/123/start`
- `POST /trips/123/finish`
- `POST /trips/123/cancel`

This is basically jamming RPC-style endpoints into a REST API, which is certainly a popular solution but is technically not REST. This crossover is a sign of how hard it can be to put actions into REST. While it might not be obvious at first, it is possible. One approach is to use a state machine, on something like a `status` field:

```
PATCH /trips/123 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"status": "in_progress"}
```

Just like any other field, you can `PATCH` the new value of `status` and have some logic in the background fire off any important actions:

```
module States
  class Trip
    include Statesman::Machine

    state :locating, initial: true
    state :in_progress
    state :complete

    transition from: :locating, to: [:in_progress]
    transition from: :in_progress, to: [:complete]

    after_transition(from: :locating, to: :in_progress) do |trip|
      start_trip(trip)
    end
```

```
    after_transition(from: :in_progress, to: :complete) do |trip|
      end_trip(trip)
    end
  end
end
```

Statesman is an incredibly simple state machine for Ruby, written by the GoCardless team. There are many other state machines in many other languages, but this is an easy one to demonstrate.

Basically, here in your controllers, `lib` code or DDD logic somewhere, you can check to see if `"status"` was passed in the `PATCH` request, and, if so, you can try to transition to it:

```
resource.transition_to!(:in_progress)
```

When this code is executed, it will either make the transition successfully and run whatever logic was defined in the `after_transition` block, or throw an error.

The success actions could be anything: sending an email, firing off a push notification, contacting another service to start watching the driver's GPS location to report where the car is — whatever you like.

There was no need for a `POST /startTrip` RPC method or a REST-ish `POST /trips/123/start` endpoint, because it could simply be handled consistently within the conventions of the REST API.

When Actions Can't Be Afterthoughts

We've seen here two approaches to fitting actions inside a REST API without breaking its RESTfulness, but depending on the type of application the API is

being built for, these approaches might start to feel less and less logical and more like jumping through hoops. One might start to wonder, Why am I trying to jam all of these actions into a REST API? An RPC API might be a great alternative, or it could be a new service to complement an existing REST API. Slack uses an RPC-based Web API, because what it's working on just would not fit into REST nicely. Imagine trying to offer "kick," "ban" or "leave" options for users to leave or be removed from a single channel or from the whole Slack team, using only REST:

```
DELETE /users/jerkface HTTP/1.1
Host: api.example.com
```

`DELETE` seems like the most appropriate HTTP method to use at first, but this request is so vague. It could mean closing the user's account entirely, which might be very different to banning the user. While it could be either of those options, it definitely would not be kick or leave. Another approach might be to try `PATCH`ing:

```
PATCH /users/jerkface HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"status": "kicked"}
```

This would be a weird thing to do, because the user's status wouldn't be globally `kicked` for everything, so it would need further information passed to it to specify a channel:

```
PATCH /users/jerkface HTTP/1.1
Host: api.example.com
Content-Type: application/json
```

```
{"status": "kicked", "kick_channel": "catgifs"}
```

Some folks try this, but this is still odd because there is a new arbitrary field being passed, and this field doesn't actually exist for the user otherwise. Giving up on that approach, we could try working with relationships:

```
DELETE /channels/catgifs/users/jerkface HTTP/1.1
Host: api.example.com
```

This is a bit better because we're no longer messing with the global `/users/jerkface` resource, but it is still missing a "kick," "ban" or "leave" option, and putting that into the body or query string is once again just adding arbitrary fields in an RPC way.

The only other approach that comes to mind is to create a `kicks` collection, a `bans` collection and a `leaves` collection, with some endpoints for `POST /kicks`, `POST /bans` and `POST /leaves` endpoints to match. These collections would allow meta data specific to the resource, like listing the channel that a user is being kicked from, for example, but it feels a lot like forcing an application into a paradigm that doesn't fit.

Slack's Web API looks like this:

```
POST /api/channels.kick HTTP/1.1
Host: slack.com
Content-Type: application/json

{
  "token": "xxxx-xxxxxxxxx-xxxx",
  "channel": "C1234567890",
  "user": "U1234567890"
}
```

Nice and easy! We're just sending arguments for the task at hand, just like you would in any programming language that has functions.

One simple rule of thumb is this:

- If an API is mostly actions, maybe it should be RPC.
- If an API is mostly CRUD and is manipulating related data, maybe it should be REST.

What if neither is a clear winner? Which approach do you pick?

Use Both REST And RPC

The idea that you need to pick one approach and have only one API is a bit of a falsehood. An application could very easily have multiple APIs or additional services that are not considered the "main" API. With any API or service that exposes HTTP endpoints, you have the choice between following the rules of REST or RPC, and maybe you would have one REST API and a few RPC services. For example, at a conference, somebody asked this question:

> *We have a REST API to manage a web hosting company. We can create new server instances and assign them to users, which works nicely, but how do we restart servers and run commands on batches of servers via the API in a RESTful way?*

There's no real way to do this that isn't horrible, other than creating a simple RPC-style service that has a `POST /restartServer` method and a `POST /execServer` method, which could be executed on servers built and maintained via the REST server.

Summary

Knowing the differences between REST and RPC can be incredibly useful

when you are planning a new API, and it can really help when you are working on features for existing APIs. It's best not to mix styles in a single API, because this could be confusing both to consumers of your API as well as to any tools that expect one set of conventions (REST, for example) and that fall over when it instead sees a different set of conventions (RPC). Use REST when it makes sense, or use RPC if it is more appropriate. Or use both and have the best of both worlds!

Front page image credit: Michel Bozgounov.

(rb, yk, al, il)