Do you really know why you prefer REST over RPC?

By Arnaud Lauret | May. 10, 2015 | Posts

A few weeks ago I've seen an interesting flock of tweets initiated by this question:

Camille Fournier
@skamille

Is my hatred of having http endpoints with the same path but different behaviors based on the verb totally irrational? Because I HATE it

25 8:09 PM - Apr 16, 2015

22 people are talking about this

This question and the tweets that followed put my brain on quite an animated discussion…

SOMEWHERE IN API HANDYMAN'S BRAIN...
HERETIC!!!!!
BUUUURN !!!!!
VOICES INSIDE MY HEAD
WHY DO I THINK RESOURCES ARE BETTER THAN OPERATION?
HAVE WE ANYTHING ABOUT THIS IN THE LIBRAINRY?
BUUUURN !!!!!
OH! SHUT UP YOU STUPID API INQUISITION!
WE CAN'T HEAR OURSELVES THINKING IN HERE!
TEELA'S REALLY SCARY!
THE API INQUISITION VALIANTLY GIVE UP THE FIELD AND RETIRE
NOPE, NOTHING ABOUT THIS IN THE LIBRAINRY
GONNA BE A TLTT EXPLAINATION...
TLTT?
TOO LONG TO TWEET. HOW COULDN'T YOU KNOW THAT? AREN'T WE SUPPOSED TO BE THE SAME PERSON!

()

After this internal discussion, I realized that this question (and all the tweet debate that follows it) could help me highlight a dark corner of my librainry: why should I considered REST's request style (resource oriented) better than RPC's (operation oriented)? Is RPC's request style so evil? Is REST's the panacea?

What RPC's and REST's requests styles look like

Before comparing the two request styles let's see what they look like.

The HTTP request

Both RPC and REST use HTTP protocol (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) which is a request/response protocol.

A basic HTTP request consists of:

- A verb (or method)
- A resource (or endpoint)

Each HTTP verb:

- Has a meaning
- Is idempotent or not: *A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request* (see RFC7231: Idempotent methods (https://tools.ietf.org/html/rfc7231#section-4.2.2)).
- Is safe or not: *Request methods are considered "safe" if their defined semantics are essentially read-only* (see RFC7231: Safe methods (https://tools.ietf.org/html/rfc7231#section-4.2.1)).
- Is cacheable or not

Verb	**Meaning**	**Idempotent**	**Safe**	**Cacheable**
GET	Reads a resource	Yes	Yes	Yes
POST	Creates a resource or triggers a data-handling process	No	No	Only cacheable if response contains explicit freshness information
PUT	Fully updates (replaces) an existing resource or create a resource	Yes	No	No

Verb	Meaning	Idempotent	Safe	Cacheable
PATCH	Partially updates a resources	No	No	Only cacheable if response contains explicit freshness information
DELETE	Deletes a resource	Yes	No	No

The table above shows only the HTTP verbs used commonly by RPC and REST APIs.

RPC: The operation request style

The RPC (https://www.acronymfinder.com/RPC.html) acronym has many meanings and Remote Procedure Call (https://en.wikipedia.org/wiki/Remote_procedure_call) has many forms.
In this post, when I talk about RPC I talk about *WYGOPIAO: What You GET Or POST Is An Operation*.

With this type of RPC, you expose *operations* to manipulate data through HTTP as a *transport protocol*.

As far as I know, there are no particular rules for this style but generally:

- The endpoint contains the name of the operation you want to invoke.
- This type of API generally only uses GET and POST HTTP verbs.

```
GET /someoperation?data=anId

POST /anotheroperation
{
  "data":"anId";
  "anotherdata":"another value"
}
```

How do people choose between GET and POST?

- For those who care a little about HTTP protocol this type of API tends to use GET for operations that don't modify anything and POST for other cases.
- For those who don't care much about HTTP protocol, this type of API tends to use GET for operations that don't need too much parameters and POST for other cases.
- Those who really don't care or who don't even think about it choose between GET and POST on a random basis or always use POST.

REST: The resource request style

I will not explain in detail what REST is, you can read Roy Fielding's dissertation (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) and The REST cookbook (http://restcookbook.com/) for more details.

To make it short and focus on the matter of this post, with a REST API you expose data as resources that you manipulate through HTTP protocol *using the right HTTP verb* :

- The endpoint contains the resource you manipulate.
- Many use the CRUD analogy to explain REST requests principles. The HTTP verb indicates what you want to do (Create/Read/Update/Delete) with that resource as defined earlier in this post and by RFC7231 (Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content) (https://tools.ietf.org/html/rfc7231#section-4.3) and RFC5789 (PATCH Method for HTTP) (https://tools.ietf.org/html/rfc5789).

```
GET /someresources/anId

PUT /someresources/anId
{"anotherdata":"another value"}
```

Examples

Here are some of my CarBoN API (/the-api-crash-test-project/) requests presented in RPC and REST ways:

Operation	**RPC (operation)**	**REST (resource)**
Signup	POST /signup	POST /persons
Resign	POST /resign	DELETE /persons/1234
Read a person	GET /readPerson?personid=1234	GET /persons/1234
Read a person’s items list	GET /readUsersItemsList?userid=1234	GET /persons/1234/items
Add an item to a person’s list	POST /addItemToUsersItemsList	POST /persons/1234/items
Update an item	POST /modifyItem	PUT /items/456
Delete an item	POST /removeItem?itemId=456	DELETE /items/456

Comparing RPC’s and REST’s requests styles

I’ve selected some items to compare RPC’s and REST’s requests styles:

- Beauty
- Designability
- API definition language
- Predictability and semantic

- Hypermediability
- Cacheability
- Usability

Beauty

Even if this item is irrelevant, as beauty is in the eye of the beholder, both styles can produce beautiful API as they can produce ugly ones.

Operation	**RPC**	**REST**
Read a person *pretty version*	GET /readPerson?personid=1234	GET /persons/1234
Read a person *ugly version*	GET /rdXbzv01?i=1234	GET /xbzv01/1234

So that's a draw for this one.

Designability

Is it easier to design RPC ou REST endpoints?

Designing a RPC API may seem easier:

- when you have to deal with an existing system as it is generally operation oriented but you'll have to simplify and clean this vision to expose it.
- when you deal mainly with processes and operations (as transform them into REST resources is not always trivial).

The design of an RPC API needs the designers to be strict to achieve a consistant API as you do not really have constraints.

Designing a REST API may seem easier when you deal mainly with data.

But even if in some certain case , designing a REST API seems a little harder than an RPC one, it gives you a *frame* that let you achieve more easily a consistent API.

And in both case you'll have to deal with naming consistency.

Both style have pros and cons depending on the context but I don't find that one style is more easier to design than the other. As *I* don't really see a winner, **that's another draw.**

API definition languages

You can perfectly describe both styles with API definition languages like Swagger, RAML or blueprint.

So that's a draw, again.

Predictability and semantic

With RPC the semantic relies (mostly) on the endpoint and there are no global shared understanding of its meaning. For example, to delete an *item* you could have:

- GET (or POST) /deleteItem?itemId=456
- GET (or POST) /removeIt?itemId=456
- GET (or POST) /trash?itemId=456

To resign from the service you could have:

- POST (or GET) /resign
- POST (or GET) /goodbye
- POST (or GET) /seeya

With RPC you rely on your human interpretation of the endpoint's meaning to understand what it does *but* you can therefore have a fine human readable description of what is happening when you call this endpoint.

With REST the semantic relies (mostly) on the HTTP verb. The verb's semantic is globally shared. The only way to delete an *item* is:

- DELETE /items/456

If a user want to stop using your service, you'll do this (not so obvious) call:

- DELETE /users/1234

REST is more predictable than RPC as it relies on the shared semantic of HTTP verbs. You don't know what happen exactly but you have a general idea of what you do.

REST wins (but shortly).

Hypermediability

In both style you end making HTTP request, so there is no problem do design an hypermedia API with any of these styles.

This is a draw.

Cacheability

I've often seen (http) caching used as a killer reason to choose REST over RPC.
But after reading HTTP RFCs, I do not agree with this argument (maybe I missed something). Of course if your RPC API only use POST for all requests, caching may be a little tricky to handle (but not impossible). If you use GET and POST wisely, your RPC API will be able to obtain the same level of cacheability as a REST API.

This is a draw.

Usability

From a developer point of view both styles are using HTTP protocol so there's basically no difference between RPC and REST request. No difference on the documentation (machine of human readable) level too.

This is a draw.

Totalling points

Item	Who wins?
Beauty	Draw
Designability	Draw
API definition language	Draw
Predictability and semantic	REST
Hypermediability	Draw
Cachability	Draw
Usability	Draw

Do REST really wins?

REST *wins* thanks to the *predictability and semantic* item.
So, is the resource approach better than the operation one?

No.

RPC and REST are only different approaches with pros and cons and both are valueable *depending on the context*. You can even mix these two approaches in a single API.

The *context*, that's the key. There are no panacea solution, don't follow fashion blindly, you always have to think within a context and must be *pragmatic* when *choosing* a solution.

At least, I know now why I *like* the resource approach: its predictability and the frame given by the full use of HTTP protocol. What about you?

One last word to leave you with food for thought: in this time of advent of functionnal programming, having operation request style could make sense…

Name

Larry Garfield • 4 years ago

I overall agree that RPC vs. REST is not a cage-match for the One True Approach(tm). However, I disagree that hypermedia-ability is a wash. For Hypermedia to even work, you need to have resources you're hyperlinking to, and those resources need to have known semantics. If you're linking to "I am a part of this collection" or "my owning user is this user", then you need to have a URI that represents that collection or user. If you have a URi that represents that collection or user, then GETing it... had really better give you some kind of information about it because I've no idea what you'd do with it otherwise.

In REST, your URIs represent NOUNS.

In RPC, your URIs represent VERBS.

A hypermedia link from noun to noun makes sense. A hypermedia link from verb to verb... I don't even know what that means.

If hypermedia links aren't relevant to your use case, then that's not relevant. But I don't think it's fair to say that hypermedia is a wash between REST and RPC generally. Hypermedia is particularly a REST/noun-based concept that has no meaning in an RPC/verb-based model.

10 ^ | ⌄ • Reply • Share ›

Michael Harris • 2 years ago

One thing I don't really understand with REST: what if there are a lot more things you need to be able to do with a resource than create, update and delete it? Or do you need to express all actions against a resource as an "Update"?

4 ^ | ⌄ • Reply • Share ›

meetmickeymoon ➦ Michael Harris • a year ago

On a similar note even for read operation, REST only suggests semantics for reading a specific instance of an entity (e.g. /users/1234) or a collection of them(e.g. /users). Do we have any semantics to filter a collection(like users with salary greater than 2000$ and age greater than 30), search, text search(products with "discount" keyword in title or description), search/filter over multiple distinct entities (like search products or merchants offering discounts by looking "discount" keyword in their description)

^ | ⌄ • Reply • Share ›

matchilling ➦ meetmickeymoon • 4 months ago • edited

Great question! What I've seen mostly in the wild are the following two approaches:

a) Create a parameter structure which allows you to query your domain resources accordingly ~ the users example is simple enough and let’s you write queries based on your model like this `GET /users?age=gt30&salary=gt2000`

b) Create a dedicated search model in your domain. If the query structure from the first example is getting more complicated and you want do let’s say for example some SQL-like queries (WHERE title = "%discount%" OR description LIKE "%discount%") I would think about introducing a separated search model

LIKE "%discount%"), I would think about introducing a separated search model which could translate into a REST call ~ e.g `GET /product_search` with some payload `{"query":{"multi_match":{"query":"discount","fields":["title","description"]}}}`.

1 ^ | ˅ 1 • Reply • Share ›

George Kremenliev • 2 years ago

I am not sure that the win in semantics is for the REST. There are a lot of cases where the CRUD is not enough. As you gave the example with "resign" but i don't think DELETE is the right verb there. If we go that rabbit hole - the act of resigning actually is an update of a property (IsResigned) but then PATCH /users/1234/HasResigned seems very contrived, especially when compared to the POST /resign?userId=1234 with some data that should go into the user's file. So in this round I would give the nod to RPC.

3 ^ | ˅ • Reply • Share ›

Kevin ➦ George Kremenliev • a year ago

In a REST system, you wouldn't do any of this. You would have /resignations that you create (POST) to initiate the process to have someone resign. Under the hood it may be as simple as updating a status column in a table, but to the clients, it would be treated like its own resource (when did they resign? who performed the resignation? all sorts of other data may be relavanat about a resignation than just a status flag).

2 ^ | ˅ • Reply • Share ›

togakangaroo • 3 years ago

I like your approach to discussing this. The only thing I can think of to add is that knowing an operation is idempotent (GET, PUT, DELETE) can enable automatic retries - especially at a proxy level. Otherwise, good discussion!

4 ^ | ˅ 1 • Reply • Share ›

Jeroen Keppens • 4 years ago

We use REST for most things that handle data: get an article, ge an overview of articles, modify an article, delete an article, etc... Commenting on an article, user registration falls under this (REST).

What we use RPC for is state changes or actions on objects that we only allow partial changes (theoretically this is a patch, but we want to avoid checking that only specific fields are being patched).

RPC (action) : GET /user/~/resendActivationMail (no update in db)
RPC (action / patch) : POST /article/123456/approve (this could be a PATCH where we set "state" to published)

REST => GET /article/123456/comments -> get comments for article 123456
=> example: one comment returned is comment 98765
REST => DELETE /comments/98765 -> hard delete of a comment
RPC => POST /comments/98765/approve -> approve a comment as administrator (instead of a patch with { "isApproved" => true, but let's say we don't allow any other part of the comment to be modified by the api consumer}

What do you think?

2 ^ | ˅ • Reply • Share ›

Dao ➦ Jeroen Keppens • 4 years ago

When you find yourself concerned with partial updates, specifically that you want to control what portions of a resource are being patched, you might consider creating resources that specifically update only those attributes. I am not looking at your domain, but in general this is a common scenario and indicates a need for more finely detailed resources (or messages about resources). You can do this RPC style or Resource style. After 30 some years i *tend* more towards either resource or message oriented styles, but only you can make that decision after understanding your domain. If the domain is simple enough sometimes a doThis() operation is all you need. It is as more systems (people and machines) interact that additional constructs are needed to deal with the complexity.

In your example it seems you have already solved this. An "approval" endpoint would work just as well as the "approve" verb. The composition and interaction of the domain entities is more important than any question of REST vs RPC vs Other for an example this simple. I personally might go with the "approval" entity as that might stand up longer as the system changes, but the isolation of the approval context is what is *most* important from my point of view.

1 ^ | ˅ • Reply • Share ›

Linh Nguyen ➦ Dao • 2 years ago

Totally agree!

^ | ˅ • Reply • Share ›

Ukaza Perdana • 9 months ago

Why not use both?

1 ^ | ˅ • Reply • Share ›

Eduardo Alcântara • 3 years ago

I like the predictability of REST, but I prefer to use RPC. What if I want to build an API that is accessible by HTTP, TCP, UDP and any other protocol I want or invent? To remove the VERB and specify a unique name for each part of the CRUD in the CRUDs, so I guess I will be far more reachable than my REST colleagues.

1 ^ | ˅ • Reply • Share ›

Omri • 4 years ago

Great post.
My 2 cents: An important point worth mentioning are the semantics of error handling.
Error handling is often a substantial part of our code.
Because of the nature of REST it is very easy to interpret HTTP errors semantically: 404 - resource does not exist, 301 - moved, etc.
By it's nature, RPC requires that you understand what you are doing in order to handle errors correctly.

1 ^ | ˅ 1 • Reply • Share ›

Tom Heijtink ➦ Omri • 3 years ago • edited

Can i assume from your comment that you are (ab)using HTTP error status codes in order to communicate domain logic constraints? This should not be the case. HTTP only describes the protocol and so do the status codes. If you are not allowed to patch only describes the protocol and so do the status codes. If you are not allowed to patch a resource so that it violates a constraint of some sort (either business, SQL or preferably both) the question is. Did the HTTP protocol do its work well? Did you receive the payload okay? Was the payload of the correct content-type? Etc. If all the questions concerning the HTTP protocol are all answered positively, your HTTP communication is successful and should return a 2xx or 3xx status. But definitely not a 4xx status. If something was wrong domain wise then you should return a body notifying the consumer what went wrong. This same rule applies to RPC API's

If one is going to pride himself in the fact that he utilizes a greater part of the HTTP protocol vocabulary by implementing a REST API then at least use the protocol as designed. And not pride yourself in one part of the standard and totally mess it up on other parts.

This part is generally something that RPC implementations get right. They return normal payload to indicate errors or warnings and leave the HTTP error status code for the HTTP protocol.

I think the whole point of the discussion is what does HTTP intend to describe. Resources or Procedures? I think that HTTP was intended to describe the former rather than the latter. This is because procedures or actions are already covered in the VERBS. And the URL of HTTP describes a path. Never the less, this doesn't mean REST is better than RPC. It's just that people have determined that an RPC architecture makes more sense to their business and decided to use the HTTP protocol to make it available. Can you blame them? Heck you also can do VPN, SSH or VNC over HTTP. This is generally used to by pass enterprise firewalls or proxies. If the ingoing HTTP(S) port is blocked they use a web socket initiated from the clients end.

2 ^ | v • Reply • Share ›

Nacho ➦ Omri • 4 years ago

Good point, but RPC can be implemented on top of HTTP too, this way you can benefit from the semantics of both, not only for error handling but also for caching, revalidation, multipart requests (i use them to upload several files in the same JSON_RPC-over-HTTP request), safety, idempotence, ...

^ | v • Reply • Share ›

Omri ➦ Nacho • 4 years ago

You are right - RPC can (and should) be implemented without breaking http standards.
But I find that when implementing REST people tend to follow HTTP semantics more strictly.
The "proprietary" nature of RPC APIs tends to lead people to misplaced "creativeness".
That is not due to any technical limitation - it is just an observation on behavior I've met in several projects.

^ | v • Reply • Share ›

Balaji Boggaram Ramanarayan • a year ago

How about 'Ease of Testing' ? REST wins

^ | v • Reply • Share ›

kevins_76 • a year ago
I'd say REST and RPC are a wash.

One slight advantage of RPC is that the same endpoint can be designed to support both GET/POST. Meaning, in this case you could use POST mostly for security and large packets. But for development and debugging (eg, tracing the network tab in Chrome) it's actually slightly easier then to use a regular web browser to access RPC url's. Everything can be accessed with GET (and/or POST).

Otherwise, curl/soapui/postman/etc can be used to troubleshoot REST calls, but it's a bit more of a hassle than a regular web browser.

REST promises to offer a more predictable interface, but it does have limitations with the HTTP abstractions. For example, one problem with using status codes for returning errors is they are singular. What happens if there's more than one error that needs to be returned? We end up putting more data in JSON, and not using the (limited) HTTP abstractions.
^ | ⌄ • Reply • Share ›

humanist1965 • 2 years ago
You talk about the HTTP GET verb being cacheable but:
- It should never be cached for a financial API (like PSD2 XS2A)
- It should not be cached when the result returned is likely to be different every-time you make the GET request.
^ | ⌄ • Reply • Share ›

WghUk • 2 years ago
Very good article!
^ | ⌄ • Reply • Share ›

Pavel Lechev • 2 years ago • edited
The whole argument of REST vs RPC is a dead-end street. Both paradigms have their sweet spots. What is far more important is choosing when to use one or the other. Also, mixing both in the same API is generally a bad idea and shows lack of forethought in API design. Seeing the same domain being manipulated by both REST- and RPC-style calls is as messy as APIs get..
^ | ⌄ • Reply • Share ›

Indhu Durai • 3 years ago
Nice article! So right about achieving cachebility in RPC.. Tunneling through multiple protocols is one other advantage for REST..
^ | ⌄ • Reply • Share ›

Wellington Torrejais • 3 years ago
Nice!
^ | ⌄ • Reply • Share ›

joruro • 3 years ago
Good post! In my opinion the RESTful it is the best approach but it isn't perfect.

I wonder how bad can be an API with an hybrid approach. For example, managing the CRUD operations with RESTful and actions (e.g login) with RPC.

I think both approaches are complementary. What is your opinion about it?

^ | ⌄ • Reply • Share ›

Arnaud Lauret Mod → joruro • 3 years ago

In the eternal fight between REST, RPC (and now GraphQL), I think that there not good or bad choice, it's always a matter of context and use case (like for every choice).
Never make arbitrary choice, what is supposed to be the "best" (because it's hype, everybody talk about it, because you like it, ...) approach may not be what is needed.
I agree about the complementarity, mixing these two different approach may sometimes be the solution.

1 ^ | ⌄ • Reply • Share ›

Binh Thanh Nguyen • 3 years ago

Thanks, nice post.

^ | ⌄ • Reply • Share ›

Erhard • 4 years ago

Intresting discussion, but keep in mind that REST is bound to http. The tendency is now also to other protocols like websockets to get bidirectional communication. Rest is made with synchronous request response cycles.

^ | ⌄ • Reply • Share ›

Katarzyna Siwek • 4 years ago

I am convinced to follow API Design Guidelines by Dix, Paul in "Service-Oriented Design with Ruby and Rails" (quote):
- URIs should be descriptive
- When possible, API endpoints should be exposed as resources
- Data should be descriptive

^ | ⌄ • Reply • Share ›

Bob • 4 years ago

The actual semantics in the application layer never mattered to me. I prefer RPC because it's called just like regular procedures/functions in code source and tends to be fast.

^ | ⌄ • Reply • Share ›

Nacho • 4 years ago

One of the major drawbacks I have experimented with REST is dealing with complex inputs. The beauty of the URLs and the resource oriented approach collapse. Imagine for example filtering a collection of resources by different categories, criteria, facets, page number, etc. This is one of the reasons that made me start working on https://github.com/brutusin....
Hopefully you find it interesting

^ | ⌄ • Reply • Share ›

Seb • 4 years ago

How about event driven architectures? How does REST fits into that world?
I assume, there a quite lot of situations where our lovely REST religion finds its boundaries.
Let's think about WebSockets, PubSub, Messaging, Enterprise Event Bus

Currently, I find google's grpc quite interesting ...

"RPC framework that puts mobile and HTTP/2 first"

http://www.grpc.io/docs/
https://medium.com/@btaylor...

^ | ⌄ • Reply • Share ›

How are these a draw? • 4 years ago

I can't wrap my head around how you can call everything a "draw", despite showing the obvious drawbacks of RPC.

Is the only way to make REST ugly by pretending that somebody built an API where they called the Person record "xbzv01"? Really?

Is RPC at its most beautiful a re-invention of HTTP's verb, but in a nonstandard place? Really?

To show off the "beauty" of the RPC style, I've put some custom text in the Website text field to tell you where the Title of my response is. Apparently that's just as good as putting my Name in the Name field and my Website in the Website field.

^ | ⌄ • Reply • Share ›

Ruediger Moeller • 5 years ago

REST works ok'ish as a client <=> server (RPC) protocol. Its abysmal as a middleware protocol because of Http (need to wait for response until next request can be sent on same connection or open myriads of connections losing request ordering). The popularity of REST IMHO is caused because its very easy to understand and a lack of understanding networking fundamentals. Using http as a middleware loses > factor of 10 compared to websocket or tcp based protocols.

^ | ⌄ • Reply • Share ›

Brian White ➦ Ruediger Moeller • 5 years ago

Which patterns work best with Websockets?

^ | ⌄ • Reply • Share ›

Seb Schmidt ➦ Brian White • 4 years ago

Thats a good question. And I have few more.

How do you implement batching of multiple api calls using REST?
What are good patterns to make a REST-API (that depends on http verbs) non-blocking?

RPC-Style API's ...
- can be easily invoked by a blocking http client AND a non-blocking UDP-Client
- can delegate responsibility to its clients to decide which network protocol fits the current context.

I have some experience with JSON-RPC 2.0 (https://en.wikipedia.org/wi....

A major difference compared to REST is the request routing.

- REST Routing : uri & http verb
- JSON-RPC Routing: uri & rpc.method

JSON-RPC ...
- is well documented

see more

2 ^ | v • Reply • Share ›

Devon Jones • 5 years ago

A point about accuracy: RPC doesn't imply HTTP. RPC has existed for a long time in many forms. CORBA, DCOM, Thrift, Avro, Protobuffs, SOAP. There are many many forms of RPC and only a fraction use HTTP as a transport mechanisim (such as SOAP).

I think you miss a huge set of points in this as a discussion, but probably the single biggest one is that a resource oriented interface provides documents that can be used, using a set of standardized semantics (http headers, standardized methods, etc) that are encoded into a huge part of the architecture of the rest of the internet. A properly designed RESTful interface should seamlessly worth with load balancers, caching, proxies, etc. With RPC, it's a crap shoot, and the details vary with. every. single. implementation.

RPC is that, it's a Procedure call. What this means is that your app that is calling the procedure is intimately coupled with the internals of the implementation. By publishing documents, a RESTful interface is making few assumptions about it's usage, resulting in a lower likelihood of breakage as that service is modified in the future. RPC based distributed systems are fundamentally more brittle over the life of the system because what's being exposed is a set of procedures, implicitly coupling the client with the logic of the service.

A couple articles that are worth reading:
http://duncan-cragg.org/blo...
http://harmful.cat-v.org/so...

^ | v • Reply • Share ›

Nicolas Grilly ➔ Devon Jones • 5 years ago

Devon,

The article is about RPC **in the context of HTTP**, not RPC in general.

You can do RPC over HTTP in a way that still benefits from load balancers, caching, proxies, etc. The trick is that URLs designate operations instead of resources.

You're assuming that coupling through operations (with RPC) is worse than coupling through data/representation (with REST). It's an old debate in computer science and to my knowledge the answer is not clear cut. It depends a lot on the problem at hand. Both approaches have their pros and cons.

In my opinion, the real power of REST is in the hypermedia, which is great when an interactive user wants to navigate in the API. But it's less useful for an automated client, which probably explains why most APIs are not really RESTful (according to Fielding's definition).

^ | v • Reply • Share ›

Devon Jones ➜ Nicolas Grilly • 5 years ago

Fair enough on the HTTP part. As for the coupling via data or coupling via operations, I'm a UNIX guy, and at least within that paradigm, the answer is pretty clear. The majority of UNIX as a problem solving tool is piping representation between operation agnostic tools. I would argue that this has created a far far less fragile and far more flexible environment for solving problems.

Coupling via operation is brittle and rigid.
Coupling via representation requires a lot of extra parsing code, and can break when representations change.

My experience is that the tradeoffs of coupling via representation are far more palatable.

^ | ˅ • Reply • Share ›

Nicolas Grilly ➜ Devon Jones • 5 years ago

I don't think your analogy between REST and piping really works.

1/ Most REST APIs are used in a request-response style. There is no multi-stage piping. You just have one client that requests a resource on a server.

2/ In the piping paradigm, the set of available commands is very large. It is not limited to the set of HTTP methods. I think the operations are very important in the piping paradigm.

^ | ˅ • Reply • Share ›

Devon Jones ➜ Nicolas Grilly • 5 years ago

That's not the connection I'm referring to, what I mean is that when using rest, you give me a document, and I'm free to parse or use that document however I like. Just as when piping between commands, the representation is the contract. You expressly pointed out that "You’re assuming that coupling through operations (with RPC) is worse than coupling through data/representation (with REST).". I'm responding to that by pointing out that making the representation your contract instead of operations is akin to operating purely on the text output from one command as the input to the next. wc doesn't care about the operations before or after it, it only cares about the text and it's own flags. It's completely neutral on what opera ions it's interacting with, because it has a defined stdin/stdout contract. This is where REST is similar - the only thing that matters is the representation passed in or coming out, it's operation and context neutral. In fact, I regularly do use rest calls in the middle of pipe chains because of this operation neutrality, something that's not as easy with rpc, because with rpc, context generally matters.

Approached another way: with rpc, you nearly always have to maintain state on both sides of the call (both server and client) to represent the status of the interaction between the systems. With a properly designed rest system, that state assumption is expressly severed.

1 ^ | ˅ • Reply • Share ›

Dave G • 5 years ago
I could be wrong but usually I hear RPC in reference to the SOAP based web services that became popular when .Net was pushing them. SOAP is very bloated and did not work well with non-Microsoft technologies and RESTful quickly surpassed as the API of choice.
• Reply • Share ›

API Handyman Dave G • 5 years ago
SOAP is a kind of RPC, it's working "very well" with non-microsoft technologies (used it with Java for years (please keep that between us ;-)).
The RPC I talk about is "not so RPC", the point is more uniform vs non-uniform interfaces (thanks goes to Darrel Miller @darrel_miller for having explaining the matter like this).
• Reply • Share ›

lesto • 5 years ago
What if you can get/delete using differenti parameter than the ID? The RPC is winner ad you can nane the parameters, with REST i nave no idea to male it clean
• Reply • Share ›

Henrik Kindvall lesto • 2 years ago
Why would you want to do a risky delete based on fields that may or may not be unique for a entity?
• Reply • Share ›

API Handyman lesto • 5 years ago
I'm not quite sure to understand the question but with REST deleting a resource is straight forward: DELETE /resources/{resourceId} for example: DELETE /items/456 will delete the item with id=456. And to get it you do GET /items/456.
• Reply • Share ›

Antanas Končius API Handyman • 5 years ago
he/she meant "what if I want to delete some resources not by id, but other field", for example how to delete user by name, not integer id, then resource identification doesn't fit into standart pattern , kind of DELETE /users/13 . in lesto's case it would be something like DELETE /users/?filter[name]=yada
• Reply • Share ›

Miguel Moquillon • 5 years ago
For me, the best approach is ... POO as coined by Alan Kay (and thus as not commonly practiced in software engineering). If we applied this approach, the endpoints identifies an object and the HTTP is the transport protocol for messaging between them. The message identifier (or name) is passed in the HTTP header and the HTTP method carries the semantic of the message:
- OPTIONS to know the messages understood by the object
- GET to request an information
- POST to trigger an operation in the behalf of the object (doesn't ask information from an object to perform a task, ask to the object to perform itself the task)
- PUT to add or update information (we don't push its new state as it is the responsibility of the

object (id est local retention))
- DELETE to remove or erase information from the object (or the object itself from the system)

^ | ⌄ • Reply • Share ›

API Handyman ➦ Miguel Moquillon • 5 years ago

I suppose you meant OOP (Object Oriented Programming).
Here's an interesting comparing OOP and REST by Mike Amundsen
http://www.devx.com/enterpr...

^ | ⌄ • Reply • Share ›

Dao Shen ➦ API Handyman • 4 years ago

Mike's analysis in that article is fantastic, but Fielding's initial solution (LWP) is muddled at best. In particular i am not at all a fan of the name/value pairs in the message and that fragile object is still embedded in there. The hints to other resources are good -- we expect Fielding to get that part right. :)

Following Amundsen's work, i like a great deal of what he is trying to lead us to. I have implemented a number of message based architectures in the past myself to lesser and greater success (pre and post REST). They do work well with objects and objected oriented analysis. Additional domain analysis (in particular understanding context -- kudos for catching that Mr. Handyman) also helps. If the messages at the boundaries of the systems are subsequently well crafted the resulting systems are less fragile, less coupled, and generally easier to reason about and to change.

Message oriented systems (and for similar reasons systems written in a purely functional style) however suffer from other problems. State cannot be wished away. It will exist either explicitly or implicitly across our systems. As others have mentioned, doing the analysis to handle this is hard work.