REST vs. GraphQL: A Critical Review

Zdenek "Z" Nemec

April 10, 2019

Are you building an API?

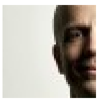
Here is the idea: If you have never heard about the REST architectural style constraints and their implication on the properties of the resulting distributed system and you do not want to (or can't) educate yourself, use GraphQL.

UPDATE

I had a **keynote** on this topic at **Nordic APIs Platform Summit 2018**. In this talk, I introduce the Framework for evaluation API Paradigms based on Constraints and Properties. I have tried to incorporate the constructive feedback on this article and provide as many unbiased insights as possible.

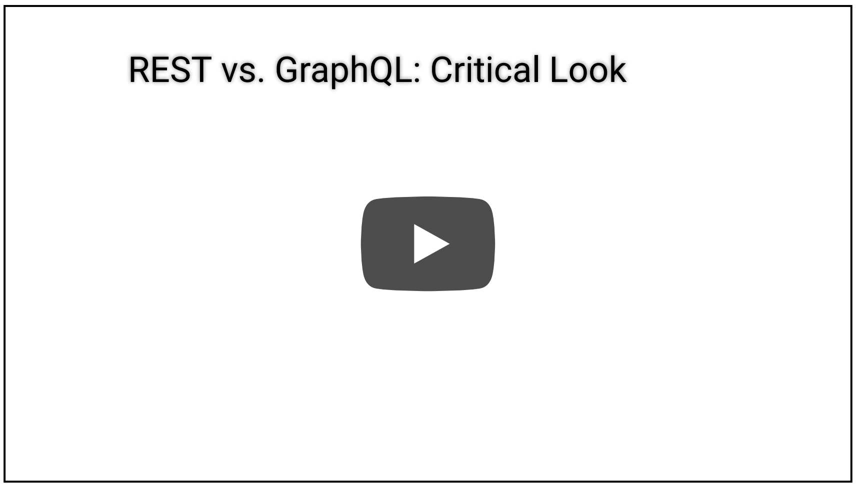
Keep in mind that is still a **critical review**, based on my experience running and working with all the different kind of APIs in production either directly or with dozens of our clients.

The deck used during the presentation is available at speakerdeck.com.

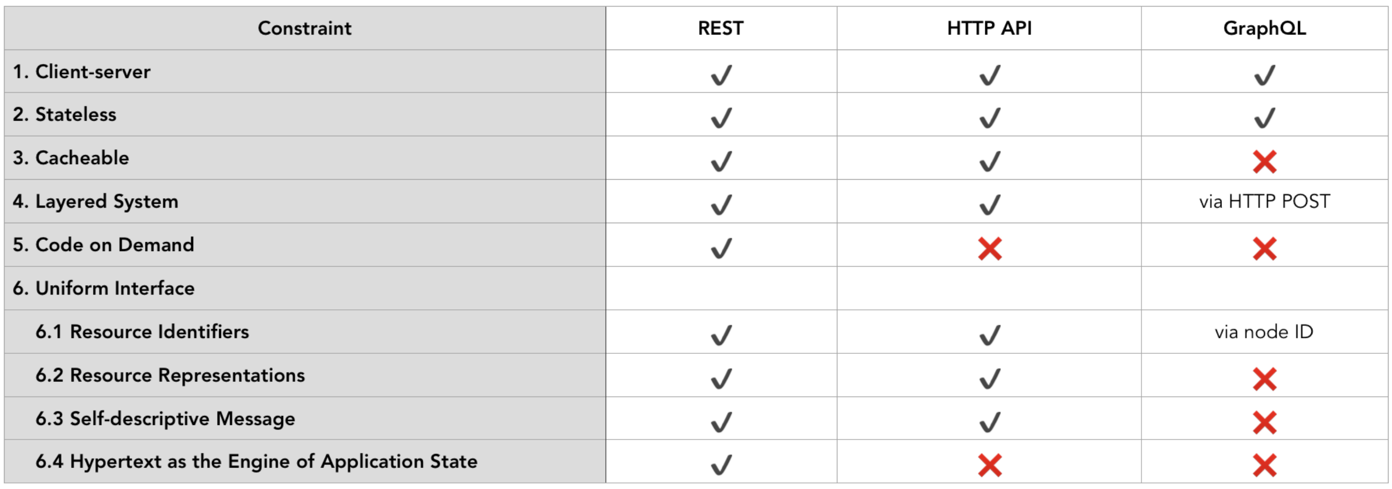


REST vs. GraphQL: Critical Review
by Z

Recording of the talk is available at YouTube:



REST vs. GraphQL: A Critical Review


REST Constraints in Popular API Techniques

Constraint	REST	HTTP API	GraphQL
1. Client-server	✓	✓	✓
2. Stateless	✓	✓	✓
3. Cacheable	✓	✓	✗
4. Layered System	✓	✓	via HTTP POST
5. Code on Demand	✓	✗	✗
6. Uniform Interface			
6.1 Resource Identifiers	✓	✓	via node ID
6.2 Resource Representations	✓	✓	✗
6.3 Self-descriptive Message	✓	✓	✗
6.4 Hypertext as the Engine of Application State	✓	✗	✗

By going with GraphQL, you will generally end up with a much better API than if you would attempt to build a REST API without understanding its concepts. After all, the lack of REST (and HTTP) knowledge resulted in the boom of “so-called-REST” APIs. And I am sure you know the problems with these APIs first hand. These problems are part of the reason for GraphQL existence.

And it makes complete sense! When time is the essence, when your API is a disposable service, when only one client that you control consumes your API, when you can’t afford to study REST or to learn HTTP in-depth or when you can’t hire someone with the expertise to help you, GraphQL might be the better way to go.

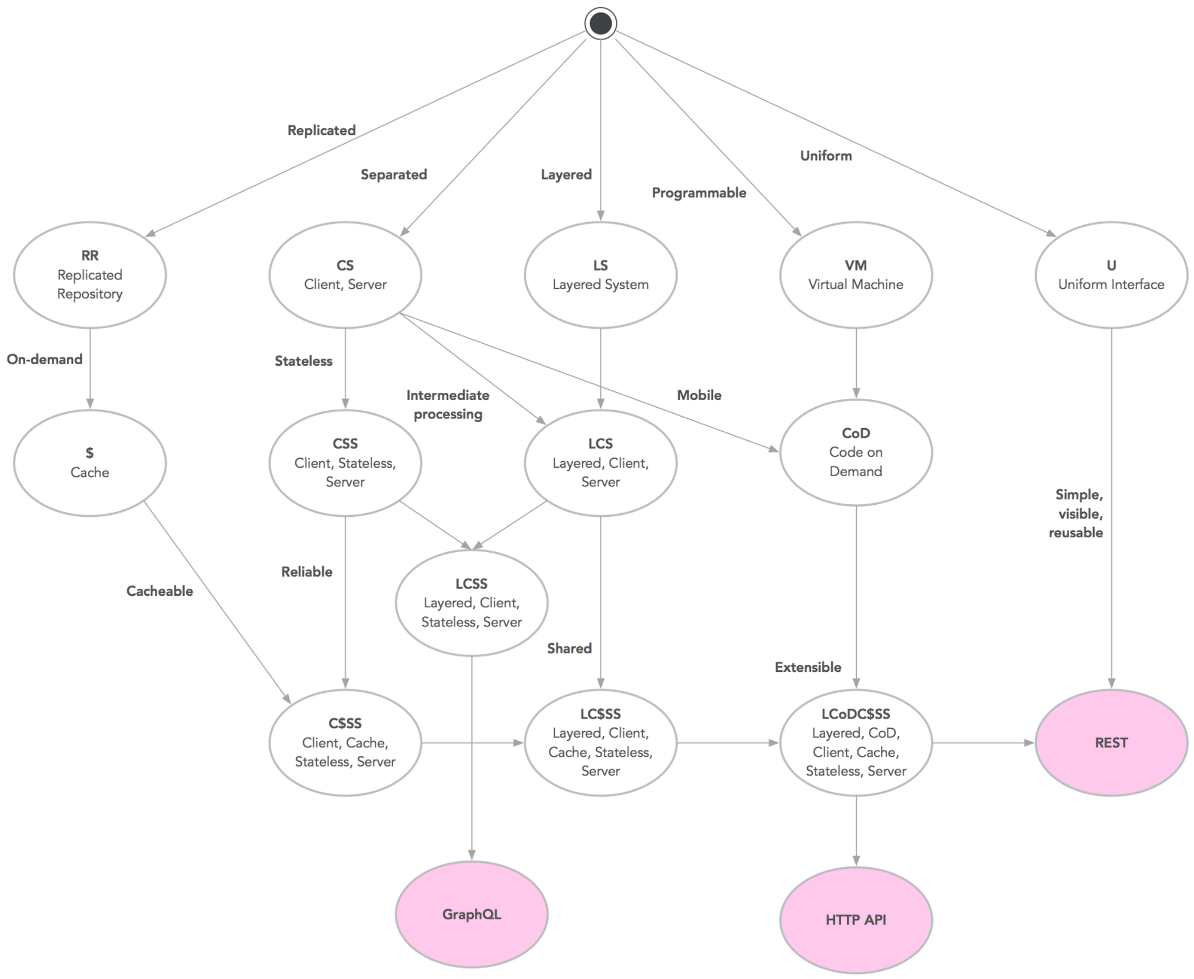
Far too many times, I’d rather see a GraphQL APIs than meet a so-called-REST API that doesn’t even understand the HTTP protocol. I am weary of trying to figure out which ID to put to which URL, or what is the difference between v2 and v3, or why do I have to use the POST HTTP method with a colossal request body to perform what is a safe, idempotent and cacheable operation.

I understand there are people who do not have the time to study REST or learn HTTP and while we can question whether they should be the ones to build APIs, with GraphQL there is a solution for them (and not only them!).

Knowledge is the Key

If you have the luxury being able to study the architectural styles or hire someone to train you, the chances are you are an architect. And as architect your role is to understand the benefits and trade-offs of different architectural styles and the implications of applying the styles constraints on the product you are building.

Of course, different products need different API architectural styles to reach the desired properties and business objectives. And as always, there is no silver bullet. As an architect, it is your duty to have multiple styles under your belt and to know which one to use and when.



Architectural styles in popular API techniques

REST APIs

True REST APIs are incredibly hard to pull off. They are so rare that outside of my ([Good API]) clients I could count them on two hands. Besides, it takes educated clients to consume the API correctly.

The reward for the martyrdom and discipline is a system that scales, performs and its components can evolve independently. But, in our fast-paced world, where many apps last only months and where we optimize for now, not later, these seem not to be the properties worth pursuing.

Beware! If you have decided to go with the microservices architecture, REST is probably the best style to enable for independent evolution of services and context separation. It almost feels like REST and microservices architecture were meant for each other.

It took me years to fully comprehend the REST, the mechanics and benefits of the server-driven application state interaction, affordance-centric design and the reality of frameworks and implementations. Depending on where you are in your API journey making a quick judgment on what you think REST is, might lead to the world of problems known as so-called-REST API.

so-called-REST APIs

There are many variants of so-called-REST APIs, that is, APIs that do not follow all the required REST constraints but still dare to label themselves as REST APIs.

The best you can hope for in this category is an API that honors the HTTP protocol and follows its semantics. I call these APIs "HTTP APIs."

A good HTTP API gives you all the benefits of the existing internet infrastructure but it still tightly couples clients with servers making an independent evolution a chore. Furthermore, it still requires a significant amount of understanding of the protocol on both producer and consumer parts.

If your HTTP knowledge isn’t the strongest and you can’t fix it, it might be a better idea to act as there is no network at all and to use GraphQL than to create an immature so-called-REST API.

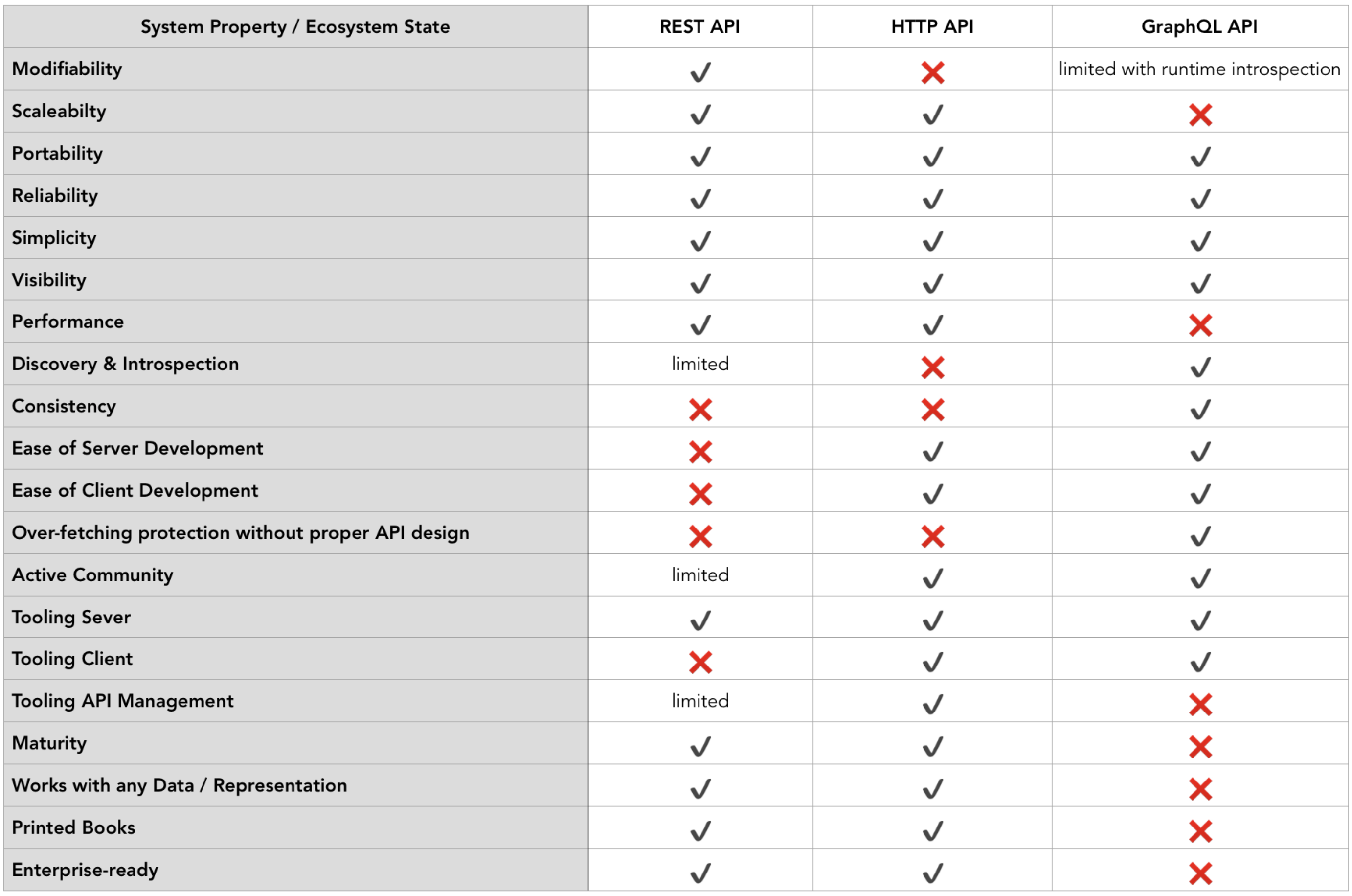
GraphQL APIs

Choosing GraphQL gives you easy to design and amazing to consume API. It also rewards you with effortless consistency between the APIs. By its nature, GraphQL is contract-driven and comes with introspection, which is something REST lacks out-of-the-box.

The cost you pay is that your clients have to read the docs (GraphQL schema) at the development time and hardcode the affordances (queries and mutations). This hardcoding of the development-time knowledge leads to clients tightly coupled to the server, but the same also applies to so-called-REST APIs.

Using GraphQL also leaves you vulnerable to what HTTP protocol and the whole internet infrastructure already solved: Scaling, performance, network communication mechanics, and concepts like content, language negotiation, and many others. Some of these can be “added” to a GraphQL API but are not included in-the-box. Adding the functionality that otherwise comes with HTTP leads to bike-shedding. And unfortunately, these home-made-solutions destroys the genius of consistency between GraphQL APIs.

Comparison of Popular API Techniques



System Property / Ecosystem State	REST API	HTTP API	GraphQL API
Modifiability	✓	✗	limited with runtime introspection
Scaleabilty	✓	✓	✗
Portability	✓	✓	✓
Reliability	✓	✓	✓
Simplicity	✓	✓	✓
Visibility	✓	✓	✓
Performance	✓	✓	✗
Discovery & Introspection	limited	✗	✓
Consistency	✗	✗	✓
Ease of Server Development	✗	✓	✓
Ease of Client Development	✗	✓	✓
Over-fetching protection without proper API design	✗	✗	✓
Active Community	limited	✓	✓
Tooling Sever	✓	✓	✓
Tooling Client	✗	✓	✓
Tooling API Management	limited	✓	✗
Maturity	✓	✓	✗
Works with any Data / Representation	✓	✓	✗
Printed Books	✓	✓	✗
Enterprise-ready	✓	✓	✗

API Design and Over-fetching

There are many misconceptions and false statements about both GraphQL and REST. The one I want to address is about the over-fetching.

Often, we see the claim that GraphQL has the benefit over REST that its clients are not over-fetching data from the server by being able to specify only the attributes they want to receive.

This statement holds true only thanks to poorly designed so-called-REST APIs.Typical examples are naive conversions of SOAP APIs into REST. Other good examples are colossal responses justified by the chattiness of the HTTP1.1 protocol. Either way, the rule of thumb is that if you see an API request or response body with more than a dozen of attributes, somebody didn’t do the API design job properly.

If you really understand your API clients use-cases (and if you use HTTP2), you can design REST API in a such a way that there will be little to no over-fetching and zero impact on the performance.

Besides, how is fetching few more fields with the out-of-the-box caching less performant than fetching only the fields you want but all the way from the origin every single time?

None Shall Escape the API Design

There’s one other aspect of understanding how users interact with your API: The API Design.

In REST you have to go through the exercise of understanding the users’ needs before the API implementation. With GraphQL you can defer the moment of understanding how users consume your API until you start profiling the queries, evaluating their complexity and identifying the slow queries.

In either case, at some point, you will have to understand the user needs and design your API and its implementation accordingly. It would be foolish to think you can create a well-performant API for every use case. You have to make design choices. There is no escaping to this.

Conclusion: Yes Man

Whether you say yes to REST, so-called-REST, or GraphQL, it is essential that you understand the consequences of the noes you tell to the other styles. What aspect are you buying when you say yes to a particular style? What other properties are you loosing when you say no to others?

Chances are you are not a technological company, and you are a product (or service) company. Take a look at the product or service you want to build, what are the business objectives? Understand your choices. Understand your decisions' impact. And don't judge by decisions of others; you are unique.

REST

pros

- Will scale indefinitely
- High performance (especially over HTTP2)

- Proven for decades
- Works with any representation
- Affordance-centric
- Server-driven application state (server tells you what you can call an when)
- Full decoupling of client and server enabling the independent evolution

neutral

- Design first, you have to think about use cases upfront, design resources and affordances accordingly.
- Multiple APIs (or dedicated resources) for specific client needs
- No reference documentation needed

cons

- Huge entry barrier in training and learning, which most of us never overcome
- The big change in the paradigm shift from SOAP, challenging for enterprises to change the mindset
- Requires clients to play along
- Poor or no tooling for clients
- No framework or tooling guidance
- Requires discipline on all sides
- You have to be an expert and then you still won’t get it right
- You do it wrong and you end up in the world of problems worse then if you would go with GraphQL
- Challenging to keep consistency and any governance

GraphQL

pros

- Easy to start with
- Easy to produce and consume
- Lots of hand-holding
- Contract-driven by nature
- Built-in introspection
- Easy to keep consistent and to govern
- Closer to SOAP which is good for enterprises

neutral

- Design later, create schema wait & evaluate what queries will users do and then optimize the queries.
- One API for all (including specific) client needs
- No reference documentation needed

cons

- Neglects the problems of the distributed system
- Server and clients coupled at the client programming time, application state is not driven by the server
- Query optimization
- Bikeshedding (content negotiation, network errors, caching), etc. etc.
- You are on your own with scaling and performance
- Throws away everything HTTP was figuring out for last 17 years
- JSON representation only
- Too few vendors in the ecosystem, the major one is pretending it owns the show

Further Reading

- Living APIs, and the Case for GraphQL
- Understanding RPC, REST and GraphQL
- Picking the right API Paradigm
- Top 5 Reasons to Use GraphQL
- API Maturity
- Reflections on the REST architectural style and “Principled design of the modern web architecture”

NOTE: GraphQL is not an API architectural style on its own. GraphQL is a query language, schema, and tooling for building RPC/tunneling APIs.

Tweet

Comments, questions?

Let's Talk!