

Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC

Two separate applications need an intermediary to talk to each other. So, developers often build bridges – [Application Programming Interfaces](#) – to allow one system to access the information or functionality of another.

In order to integrate applications quickly and at scale, APIs are realized using protocols and/or specifications to define the semantics and syntax of the messages passed across the wire. These specifications make up the API architecture.

Over time, different API architectural styles have been released. Each of them has its own patterns of standardizing data exchange. A pull of choices raises endless debates as to which architectural style is best.




API styles over time, Source: [Rob Crowley](#)

Today, many API consumers refer to REST as “*REST in peace*” and cheer for GraphQL, while ten years ago it was a reverse story with REST as the winner to replace SOAP. The problem with these opinions is that they are one-sided picking a technology itself instead of considering how its actual properties and characteristics match the situation at hand.

In this article, we'll stay objective and discuss the four major API styles in the order of their appearance, compare their strong and weak sides, and highlight the scenarios where each of them fits best.

| API ARCHITECTURAL STYLES | | | | |
|--------------------------|---|---|---|--|
| | RPC | SOAP | REST | GraphQL |
| Organized in terms of | local procedure calling | enveloped message structure | compliance with six architectural constraints | schema & type system |
| Format | JSON, XML, Protobuf, Thrift, FlatBuffers | XML only | XML, JSON, HTML, plain text, | JSON |
| Learning curve | Easy | Difficult | Easy | Medium |
| Community | Large | Small | Large | Growing |
| Use cases | Command and action-oriented APIs; internal high performance communication in massive micro-services systems | Payment gateways, identity management CRM solutions financial and telecommunication services, legacy system support | Public APIs simple resource-driven apps | Mobile APIs, complex systems, micro-services |

 **altexsoft**
software r&d engineering

Four major API styles compared

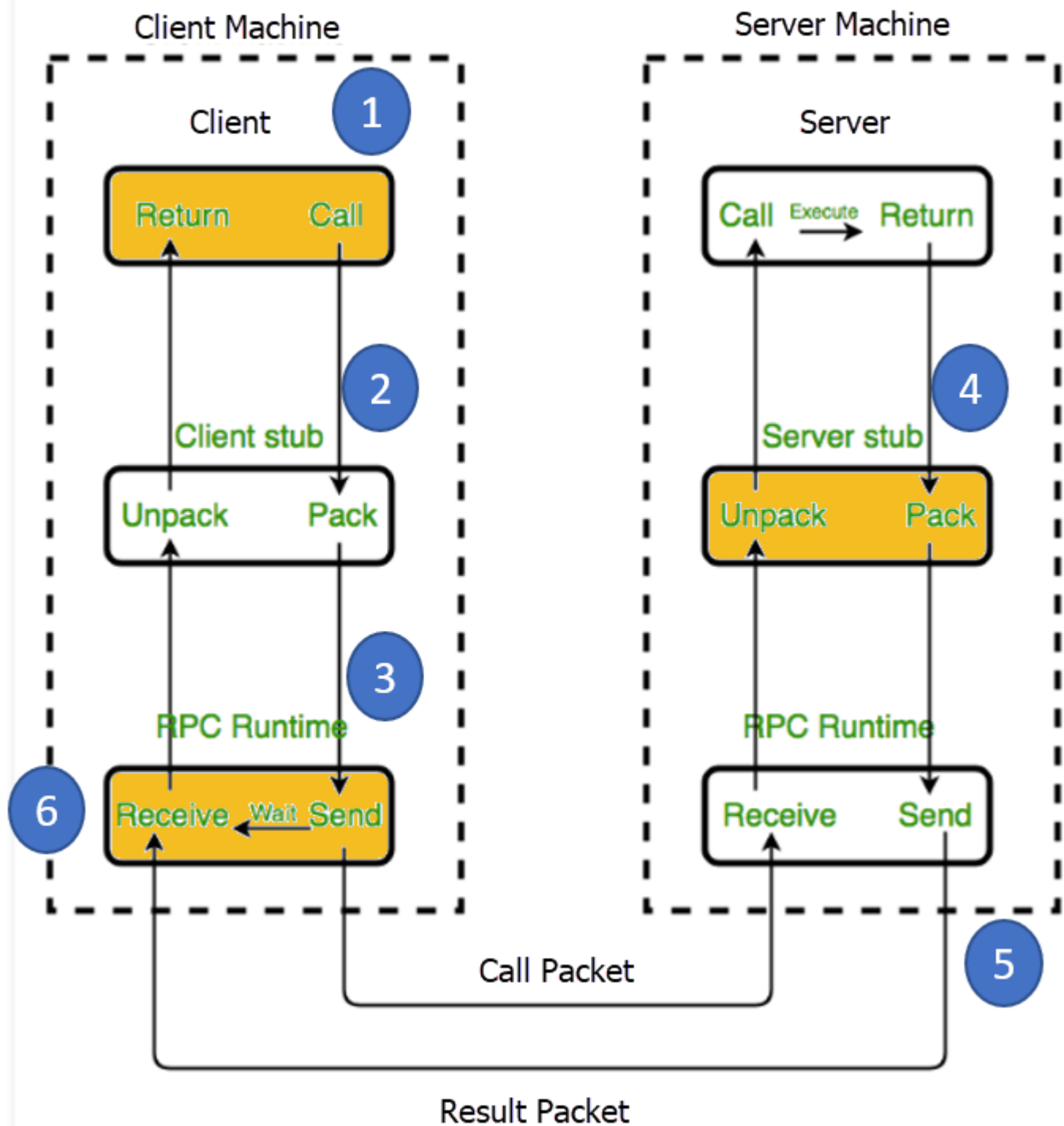
Remote Procedure Call (RPC): invoking a function on another system

A **Remote Procedure Call** is a specification that allows for remote execution of a function in a different context. RPC extends the notion of local procedure calling but puts it in the context of an HTTP API.

Initial XML-RPC was problematic because ensuring data types of XML payloads is tough. So, later an RPC API started using a more concrete [JSON-RPC](#) specification which is considered a simpler alternative to SOAP. [gRPC](#) is the latest RPC version developed by Google in 2015. With pluggable support for load balancing, tracing, health checking, and authentication, gRPC is well-suited for connecting microservices.

How RPC works

A client invokes a remote procedure, serializes the parameters and additional information into a message, and sends the message to a server. On receiving the message, the server deserializes its content, executes the requested operation, and sends a result back to the client. The server stub and client stub take care of the serialization and deserialization of the parameters.



Remote Procedure Calling Mechanism, Source: [Guru99](#)

RPC Pros

Straightforward and simple interaction. RPC uses GET to fetch information and POST for everything else. The mechanics of the interaction between a server and a client come down to calling an endpoint and getting a response.

Easy-to-add functions. If we get a new requirement for our API, we can easily add another endpoint executing this requirement: 1) Write a new function and throw it behind an endpoint and 2) now a client can hit this endpoint and get the info meeting the set requirement.

High performance. Lightweight payloads go easy on the network providing high performance, which is important for shared servers and for parallel computations executing on networks of workstations. RPC is able to optimize the network layer and make it very efficient with sending tons of messages per day between different services.

RPC Cons

Tight coupling to the underlying system. An API's abstraction level contributes to its reusability. The tighter it is to the underlying system, the less reusable it will be for other systems. RPC's tight coupling to the underlying system doesn't allow for an abstraction layer between the functions in the system and the external API. This raises security issues as it's quite easy to leak implementation details about the underlying system into the API. An RPC's tight coupling makes scalability requirements and loosely coupled teams hard to achieve. So, the client either worries about any possible side effects of calling a particular endpoint or tries figuring out what endpoint to call because it doesn't understand how the server is naming its functions.

Low discoverability. In RPC there's no way to introspect the API or send a request and start understanding what function to call based on its requests.

Function explosion. It's so easy to create new functions. So, instead of editing the existing ones, we create new ones ending up with a huge list of overlapping functions that are hard to understand.

RPC use cases

The RPC pattern started being used around the 80s, but this doesn't automatically make it obsolete. Big companies like Google, Facebook ([Apache Thrift](#)), and Twitch ([Twirp](#)) are using RPC high-performance variates internally to perform extremely high-performance, low-overhead messaging. Their massive microservices systems require internal communication to be clear while arranged in short messages.

Command API. An RPC is the proper choice for sending commands to a remote system. For instance, a Slack API is very command-focused: Join a channel, leave a channel, send a message. So, the designers of the Slack API modeled it in an RPC-like style making it small, tight, and easy to use.

Customer-specific APIs for internal microservices. Having direct integration between a single provider and consumer, we don't want to spend a lot of time transmitting a lot of metadata over the wire, like a REST API does. With high message rate and message performance, gRPC and Twirp are strong cases for microservices. Using HTTP 2 under the hood, gRPC is able to optimize the network layer and

make it very efficient with sending tons of messages per day between different services. However, if you're not aiming at high network performance, but rather at a stable API contact between teams publishing highly distinctive microservices, REST will ensure that.

Simple Objects Access Protocol (SOAP): making data available as services

SOAP is an XML-formatted, highly standardized web communication protocol. Released by Microsoft a year after XML-RPC, SOAP inherited a lot from it. When REST followed, they were first used in parallel, but soon REST won the popularity contest.

How SOAP works

XML data format drags behind a lot of formality. Paired with the massive message structure, it makes SOAP the most verbose API style.

A SOAP message is composed of:

- an envelope tag that begins and ends every message,
 - a body containing the request or response
 - a header if a message must determine any specifics or extra requirements, and
 - a fault informing of any errors that can occur throughout the request processing.
-

```

<?xml version='1.0' Encoding='UTF-8' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
      <m:reference>uuid:093a2da1-q345-7391-ba5d-pqff98fe8j7d</m:reference>
      <m:dateAndTime>2007-11-29T13:20:00.000-05:00</m:dateAndTime>
    </m:reservation>
    <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
      <n:name>Fred Bloggs</n:name>
    </n:passenger>
  </env:Header>
  <env:Body>
    <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
      <p:departure>
        <p:departing>New York</p:departing>
        <p:arriving>Los Angeles</p:arriving>
        <p:departureDate>2007-12-14</p:departureDate>
        <p:departureTime>late afternoon</p:departureTime>
        <p:seatPreference>aisle</p:seatPreference>
      </p:departure>
      <p:return>
        <p:departing>Los Angeles</p:departing>
        <p:arriving>New York</p:arriving>
        <p:departureDate>2007-12-20</p:departureDate>
        <p:departureTime>mid-morning</p:departureTime>
        <p:seatPreference></p:seatPreference>
      </p:return>
    </p:itinerary>
  </env:Body>
</env:Envelope>

```

An example of the SOAP message. Source: [IBM](#)

The SOAP API logic is written in Web Service Description Language (WSDL). This API description language defines the endpoints and describes all processes that can be performed. This allows different programming languages and IDEs to quickly set up communication.

SOAP supports both stateful and stateless messaging. In a stateful scenario, the server stores the received information that can be really heavy. But it's justified for operations involving multiple parties and complex transactions.

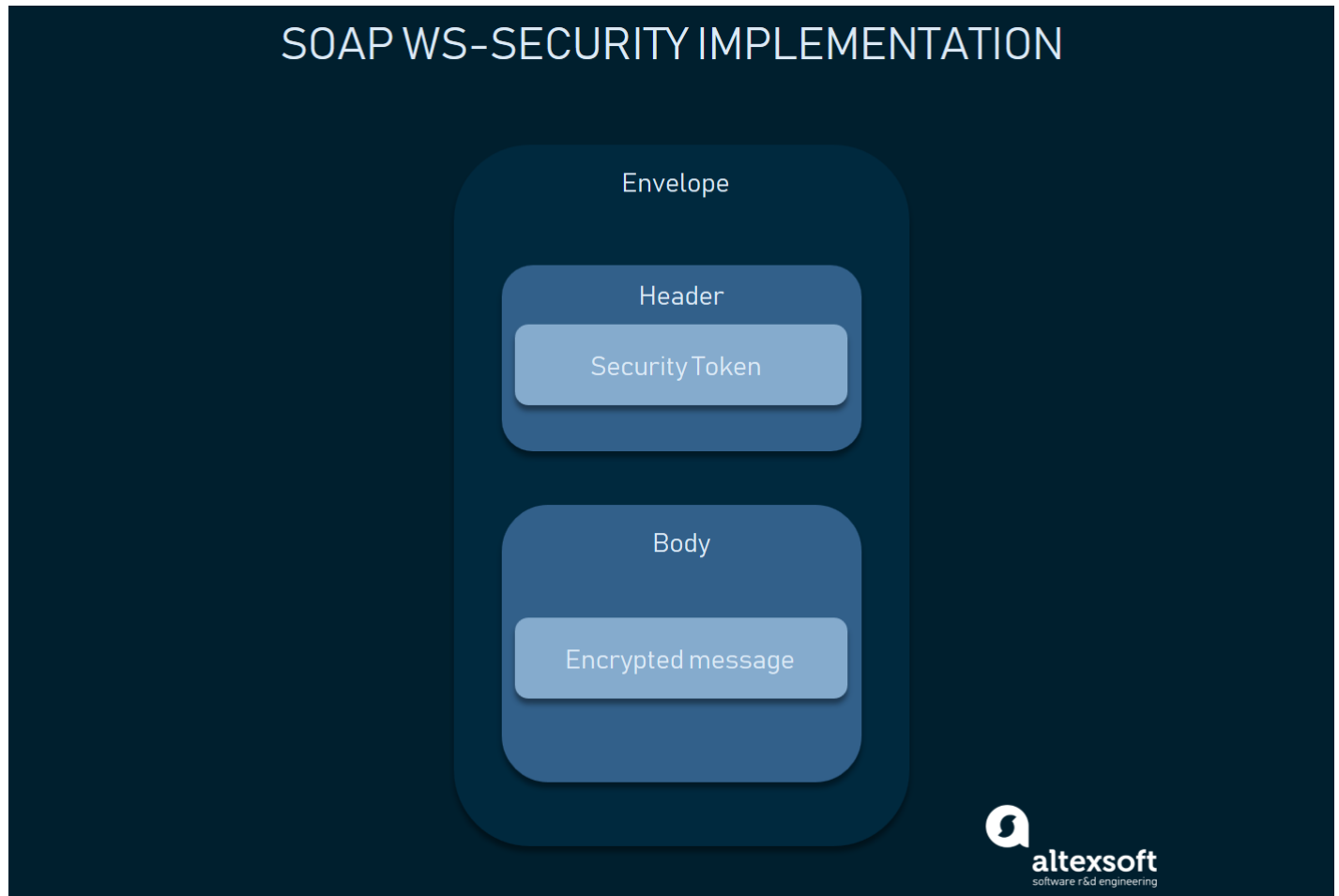
SOAP pros

Language- and platform-agnostic. The built-in functionality to create web-based services allows SOAP to handle communications and make responses language- and platform-independent.

Bound to a variety of transport protocols. SOAP is flexible in terms of transfer protocols to accommodate for multiple scenarios.

Built-in error handling. SOAP API specification allows for returning the Retry XML message with error code and its explanation.

A number of security extensions. Integrated with the WS-Security protocols, SOAP meets an enterprise-grade transaction quality. It provides privacy and integrity inside the transactions while allowing for encryption on the message level.



SOAP message-level security: authentication data in the header element and encrypted body

SOAP cons

These days, many developers shudder at the idea of having to integrate a SOAP API for several reasons.

XML only. SOAP messages contain a lot of metadata and only support verbose XML structures for requests and responses.

Heavyweight. Due to the large size of XML-files, SOAP services require a large bandwidth.

Narrowly specialized knowledge. Building SOAP API servers requires a deep understanding of all protocols involved and their highly restricted rules.

Tedious message updating. Requiring additional effort to add or remove the message properties, rigid SOAP schema slows down adoption.

SOAP use cases

Right now, the SOAP architecture is most commonly used for internal integration within enterprises or with their trusted partners.

Highly secured data transmission. SOAP rigid structure, security and authorization capabilities make it the most suitable option for enforcing a formal software contract between API and client while complying with the legal contract between the API provider and API consumer. That's why financial organizations and other corporate users opt for SOAP.

Representational state transfer (REST): making data available as resources

REST is a self-explanatory API architectural style defined by a set of architectural constraints and intended for wide adoption with many API consumers.

The most common API style today was originally described in 2000 by Roy Fielding in his [doctoral dissertation](#). REST makes server-side data available representing it in simple formats, often JSON and XML.

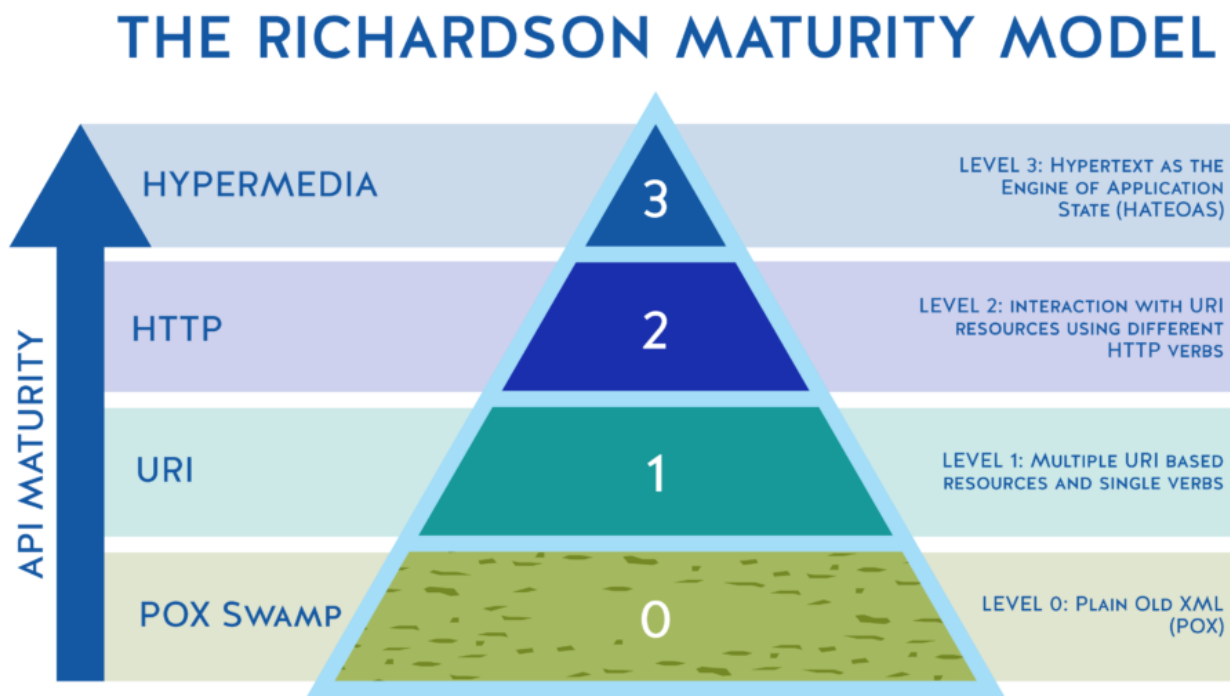
How REST works

REST isn't as strictly defined as SOAP. RESTful architecture should comply with six architectural constraints:

- **uniform interface:** permitting a uniform way of interacting with a given server regardless of device or application type
- **stateless:** the necessary state to handle the request as contained within the request itself and without the server storing anything related to the session
- **caching**
- **client-server architecture:** allowing for independent evolution of either side
- **layered system** of the application
- the ability for servers to **provide executable code** to the client

In fact, some services are RESTful only to a degree. They have RPC style at the core, break down larger services into resources, and use HTTP infrastructure efficiently. But the key part is using hypermedia aka

HATEOAS, short for [Hypertext As The Engine of Application State](#). Basically, it means that with each response, a REST API provides metadata linking to all the related info about how to use the API. That's what enables decoupling the client and the server. As a result, both API provider and API consumer can evolve independently without hindering their communication.



Richardson Maturity Model as a goalpost to achieving truly complete and useful APIs, Source: [Kristopher Sandoval](#)

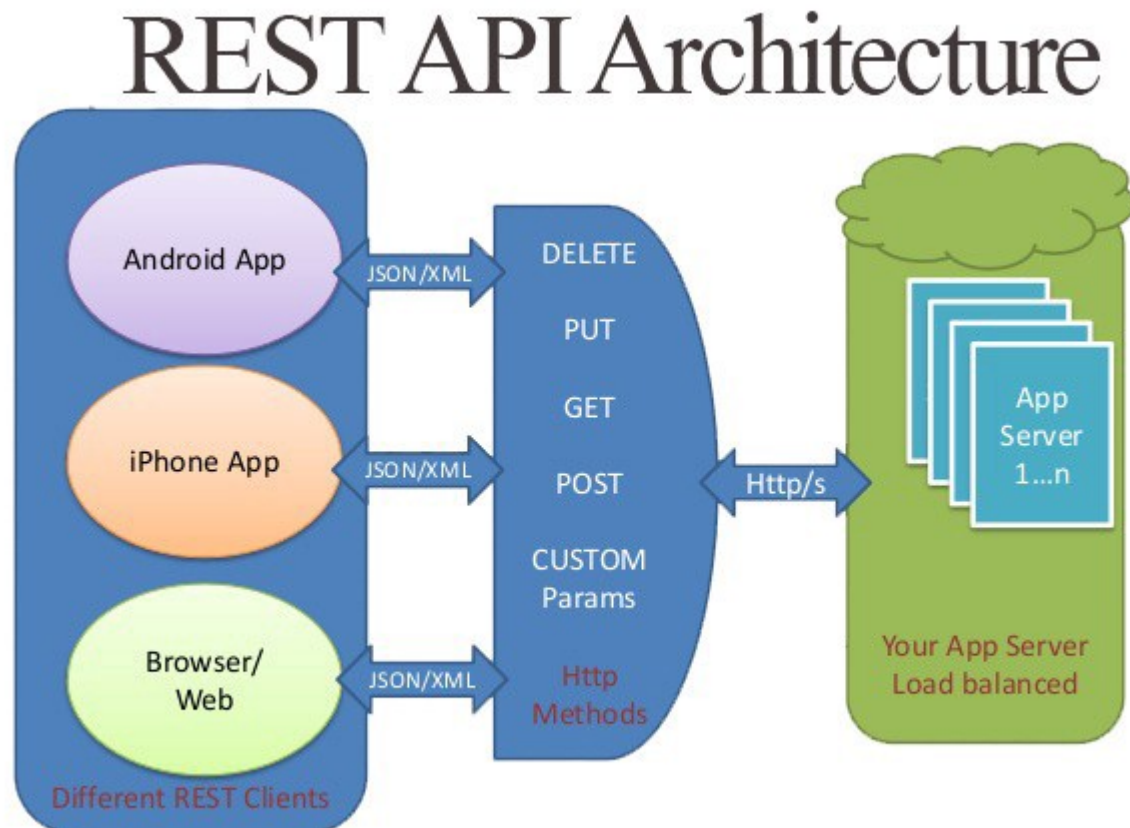
"HATEOAS is a key feature of REST. It's really what makes REST REST. Since most people don't use HATEOAS, they are actually using HTTP RPC," that's some radical opinion expressed on [Reddit](#). Indeed, HATEOAS is the most mature version of REST. However, it's difficult to achieve requiring much more advanced and intelligent API clients than those typically used and built today. So, even really good REST APIs today don't always do it. This is why HATEOAS mainly serves as a vision for the long term development of a RESTful API design.

There really may be a gray zone between REST and RPC, when a service implements some features of REST and some of RPC. REST is based on the resource or noun instead of action or verb-based.

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

Opposing operations in verb-centric RPC to the ones in noun-centric REST

In REST, things are done using HTTP methods such as GET, POST, PUT, DELETE, OPTIONS, and, hopefully, PATCH.



REST pros

Decoupled client and server. Decoupling the client and the server as much as possible, REST allows for a better abstraction than RPC. A system with abstraction levels is able to encapsulate its details to better identify and sustain its properties. This makes a REST API flexible enough to evolve over time while remaining a stable system.

Discoverability. Communication between the client and server describes everything so that no external documentation is required to understand how to interact with the REST API.

Cache-friendly. Reusing a lot of HTTP tools, REST is the only style that allows caching data on the HTTP level. In contrast, caching implementation on any other API will require configuring an additional cache module.

Multiple formats support. The ability to support multiple formats for storing and exchanging data is one of the reasons REST is currently a prevailing choice for building public APIs.

REST cons:

No single REST structure. There's no exact right way to build a REST API. How to model resources and which resources to model will depend on each scenario. This makes REST simple in theory, but difficult in practice.

Big payloads. REST returns a lot of rich metadata so that the client can understand everything necessary about the state of the application just from its responses. And this chattiness is no big deal for a big network pipe with lots of bandwidth capacity. But that's not always the case. This was the key driving factor for Facebook coming up with the description of GraphQL style in 2012.

Over- and under-fetching problems. Containing either too much data or not enough of it, REST responses often create the need for another request.

REST use cases

Management APIs. APIs focused on managing objects in a system and intended for many consumers are the most common API type. REST helps such APIs to have strong discoverability, good documentation, and it fits this object model well.

Simple resource-driven apps. REST is a valuable approach for connecting resource-driven apps that don't need flexibility in queries.

GraphQL: querying just the needed data

It takes a number of calls to the REST API for it to return the needed stuff. So GraphQL was invented to be a game-changer.

GraphQL is a syntax that describes how to make a precise data request. Implementing GraphQL is worth it for an application's data model with a lot of complex entities referencing each other.



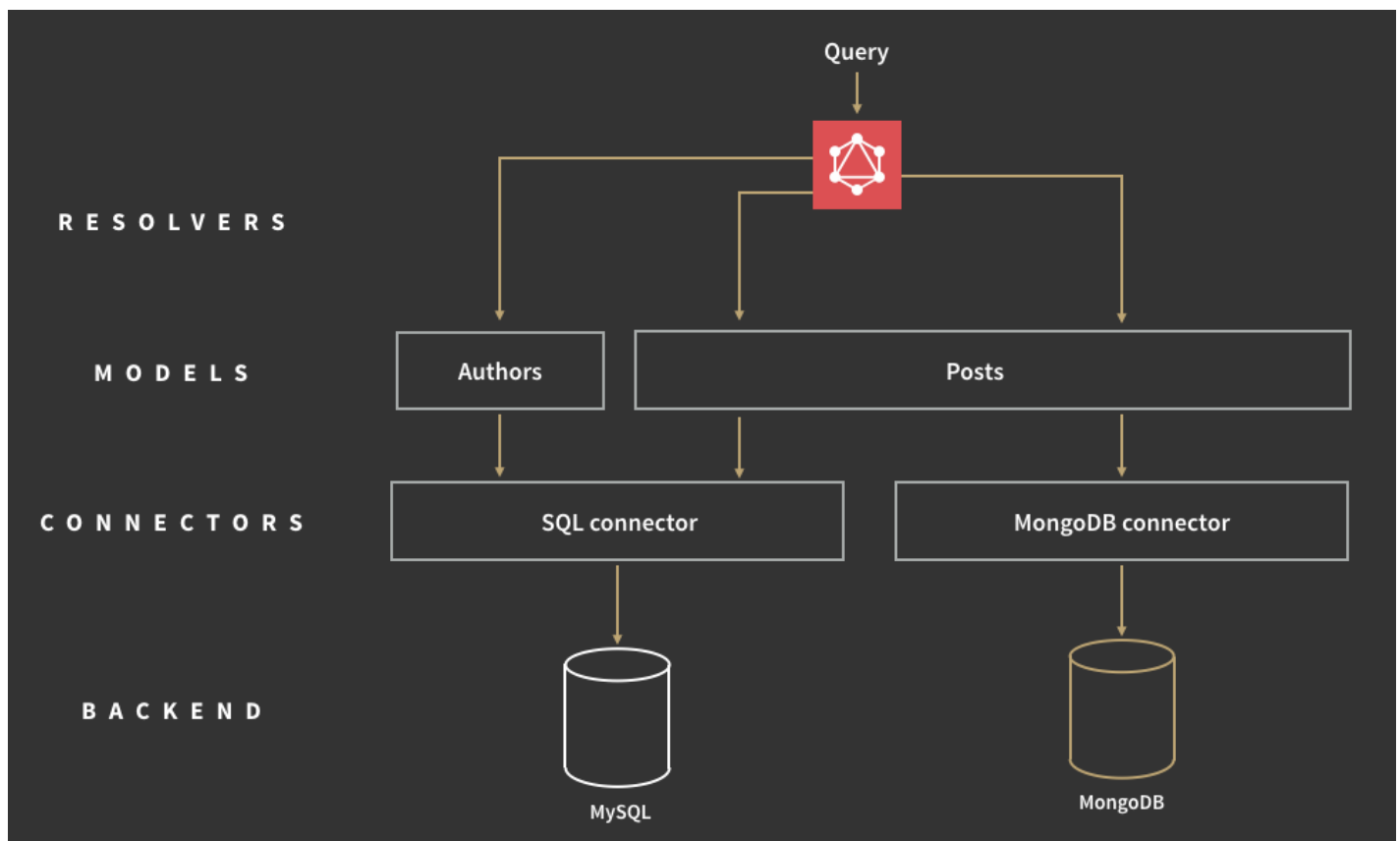
How to retrieve only the needed data from the GraphQL endpoint, Source: [Mohit Tikoo](#)

These days the GraphQL ecosystem is expanding with libraries and powerful tools like Apollo, GraphQL, and GraphQL Explorer.

How GraphQL works

GraphQL starts with building a *schema*, which is a description of all the queries you can possibly make in a GraphQL API and all the *types* that they return. Schema-building is hard as it requires strong typing in the Schema Definition Language (SDL).

Having the schema before querying, a client can validate their query against making sure the server will be able to respond to it. On reaching the backend application, a GraphQL operation is interpreted against the entire schema, and resolved with data for the frontend application. Sending one massive query to the server, the API returns a JSON response with exactly the shape of the data we asked for.



Query execution in GraphQL, Source: [Jonas Helfer](#)

In addition to the RESTful CRUD operations, GraphQL has *subscriptions* allowing for real-time notifications from the server.

GraphQL pros

Typed schema. GraphQL publishes in advance what it can do, which improves its discoverability. By pointing a client at the GraphQL API, we can find out what queries are available.

Fits graph-like data very well. Data that goes far into linked relations but not good for flat data.

No versioning. The best practice with versioning is not to version the API at all.

While REST offers multiple API versions, GraphQL uses a single, evolving version that gives continuous access to new features and contributes to cleaner, more maintainable server code.

Detailed error messages. In a similar fashion to SOAP, GraphQL provides details to errors that occurred. Its error message includes all the resolvers and refers to the exact query part at fault.

Flexible permissions. GraphQL allows for selectively exposing certain functions while preserving private information. Meanwhile, REST architecture doesn't reveal data in portions. It's either all or nothing.

GraphQL cons

Performance issues. GraphQL trades off complexity for its power. Having too many nested fields in one request can lead to system overload. So, REST remains a better option for complex queries.

Caching complexity. As GraphQL isn't reusing HTTP caching semantics, it requires a custom caching effort.

A lot of pre-development education. Not having enough time to figure out GraphQL niche operations and SDL, many projects decide to follow the well-known path of REST.

GraphQL use cases

Mobile API. In this case, network performance and single message payload optimization is important. So, GraphQL offers a more efficient data loading for mobile devices.

Complex systems and microservices. GraphQL is able to hide the complexity of multiple systems integration behind its API. Aggregating data from multiple places, it merges them into one global schema. This is particularly relevant for [legacy infrastructures](#) or third-party APIs that have expanded over time.

Which API pattern fits your use case best?

Every API project has different requirements and needs. Usually, the architectural choice depends on

- the programming language in use,
- the environment in which you're developing, and
- the resources you have to spare, both human and financial.

Knowing all the tradeoffs that go into each design style, API designers can pick the one that's going to fit the project best.

With its tight coupling, RPC works for internal microservices but it's not an option for a strong external API or an API service.

SOAP is troublesome but its rich security features remain irreplaceable for billing operations, booking systems, and payments.

REST has the highest abstraction and best modeling of the API. But it tends to be heavier on the wire and chattier – a downside if you're working on mobile.

GraphQL is a big step forward in terms of data fetching but not everyone has enough time and effort to get the hang of it.

At the end of the day, it makes sense to try a few small use cases with a particular style, and see if it fits your use case and solves your problems. If it does, try expanding and see if it fits more use cases.