A COMPARATIVE ANALYSIS OF DIFFERENT API ARCHITECTURES

Sunayana Hubli

ABSTRACT: The usage of APIs has grown consistently over the past years, as businesses realize the growth opportunities associated with running an open platform. API is a collection of open protocols and standards used for exchanging data between applications or systems. APIs are the building blocks that allow interoperability for major business platforms on the web. Software applications written in various programming languages and running on various platforms can use APIs to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. During the design of web product that is bound to use web services, the question of choosing API architecture style rises. Different APIs like RPC, SOAP (Simple Object Access Protocol), REST (Representational State Transfer) and GraphQL are some of the popular API architectures. There isn't one API style that outperforms the other in every scenario, each API has different advantages and disadvantages. In this paper we have studied and described the differences and best practices of different API architectures. This comparative analysis can aid and assist in choosing the right API architecture when designing solutions on the web.

Keywords: SOAP, REST, GraphQL, RPC, XML, HTML, Client/Server

INTRODUCTION

As businesses and developers are investing their time and resources to add dynamic systems to their apps, the importance of web APIs (Application Programming Interface) has increased hugely in building dynamic enterprise-grade applications. Hence, the choice of the API architecture is crucial to ensure a robust and efficient solution. One of the biggest challenges for an API developer when building an app with API is choosing the one that can be used for a long time. Over time, different API architectural styles have been released. Each of them has its own patterns of

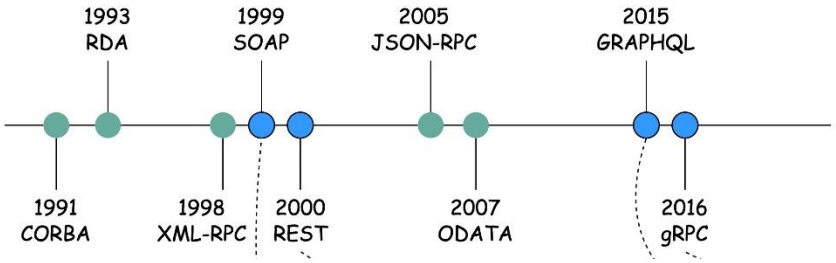
standardizing data exchange. Multiple options to choose from raises endless debates as to which architectural style is best.

Remote Procedure Call (RPC) is one of the simplest API paradigms, in which a client executes a block of code on another server. Whereas REST is about resources, RPC is about actions. Clients typically pass a method name and arguments to a server and receive back JSON or XML.[1]

SOAP is designed to be a lightweight, platform independent protocol for decentralized, distributed environment which uses Internet and XML to exchange information between nodes. It represents a messaging protocol, which uses XML to define the communication and HTTP to transmit these messages. It is a stateless, one-way message communication between nodes or devices, from sender to receiver. [2]

REST represents a client-server architecture where the client sends the requests, while the server processes them and returns the responses. It was introduced in 2000, by Roy Fielding. Unlike SOAP, REST services does not limit itself to XML, instead, it also supports JSON (JavaScript Object Notation), plain text, etc [3]

GraphQL is a query language for APIs that has gained significant traction recently. Unlike REST and RPC APIs, GraphQL APIs need only a single URL endpoint. Similarly, you do not need different HTTP verbs to describe the operation. Instead, you indicate in the JSON body whether you’re performing a query or a mutation. [1]



[5] API styles timeline

PROBLEM DEFINITION

From a technical perspective, designing an API is relatively easy. But, designing one that is maintainable, efficient, and scalable for the solution, is crucial. Many a times, API style chosen without considering the pros and cons leads to redesign, performance issues and limited functionality (scalability issues). In this Term paper, a comparison will be made on different API Architectures that are widely used. This comparative analysis can aid and assist in choosing the right API architecture when designing solutions on the web.

METHODS AND MATERIALS

API ARCHITECTURES

1. 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.

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.

2. **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.SOAP supports both stateful and stateless messaging.

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

4. GraphQL: querying just the needed data

It takes a number of calls to the REST API for it to return the needed staff. 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. These days the GraphQL ecosystem is expanding with libraries and powerful tools like Apollo, GraphiQL, and GraphQL Explorer.[7]

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.

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

ADVANTAGES AND DISADVANTAGES

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.

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..
- 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.

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.

Cons

- 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. [8]
- Tedious message updating. Requiring additional effort to add or remove the message properties, rigid SOAP schema slows down adoption.

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
- 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.
- 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.

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.. 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.

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.

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.

USE CASES AND APPLICATIONS

RPC

- 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. One notable example of an RPC-style web API is Slack API.[1]
  Following example demonstrates an example of a POST request to Slack's conversations.archive RPC API.

```
POST /api/conversations.archive
HOST slack.com
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer xoxp-1650112-jgc2asDae
channel=C01234
```

  Slack Conversations API (Figure 2-1) allows several actions, like archive, join, kick, leave, and rename. Although in this case there is a clear "resource," not all of these actions would fit into the REST pattern nicely. Additionally, there are other actions, such as posting a message with chat.postMessage, which have complex relationships with message resources, attachment resources, and visibility settings within the web client.

- 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. Most services at Twitter are built on Finagle, a protocol-agnostic and asynchronous RPC system that allows teams to build robust clients and servers. Finagle has a lot of layers, each layer implementing some portion of this RPC system that seamlessly handles heterogeneity, failures, and concurrency – all characteristics of a distributed system.[6]

REST

- 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 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.

CONCLUSION

Each API style has many pros and cons as described in this paper along with use cases. 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.

	RPC	SOAP	REST	GraphQL
API feature	Exposes action-based API methods—clients pass method name and arguments	XML-formatted, highly standardized web communication protocol	Exposes data as resources and uses standard HTTP methods to represent CRUD operations	A query language for APIs—clients define the structure of the response

Example services	Slack, Flickr, Google, Facebook (Apache Thrift), and Twitch (Twirp)	NetSuite, Salesforce	Stripe, GitHub, Google	Facebook, GitHub, Yelp
Syntax	GET /users.get?id=<*id*>	<SOAP-ENV:Envelope xmlns:SOAP-ENV = “” SOAP-ENV:encodingStyle =”” <SOAP-ENV:Header> .. </SOAP-ENV:Header> <SOAP-ENV:Body> <SOAP-ENV:Fault>	GET /users/<*id*>	query ($id: String!) { user(login: $id) { name company createdAt } }
Community	Large	Small	Large	Growing
HTTP verbs used	GET, POST	HTTP POST	GET, POST, PUT, PATCH, DELETE	GET, POST
Pros	•Easy to understand • Lightweight payloads • High performance	• Language- and platform agnostic. •Bound to a variety of transport protocols. •Built-in error handling	• Standard method name, arguments format, and status codes • Utilizes HTTP features • Easy to maintain	• Saves multiple round trips • Avoids versioning • Smaller payload size • Strongly typed • Built-in introspection
Cons	• Discovery is difficult • Limited standardization • Can lead to function explosion	•XML only. •Heavyweight. •Narrowly specialized knowledge. •Tedious message updating.	• Big payloads • Multiple HTTP round trips	• Requires additional query parsing • Backend performance optimization is difficult • Too complicated for a simple API
Learning Curve	Easy	Difficult	Easy	Medium
Format	JSON, XML, Protobuf, Thrift, FlatBuffers	XML Only	XML, JSON, HTML, plain text	JSON

- Table 1: API styles comparison summary

REFERENCES/BIBLIOGRAPHY

[1] “Designing Web APIs”, September 2018: First Edition, Brenda Jin, Saurabh Sahni, and Amir Shevat
[2] Ranga, Virender & Soni, Anshu. (2019). API Features Individualizing of Web Services: REST and SOAP. International Journal of Innovative Technology and Exploring Engineering. 8.10.35940/ijitee.I1107.0789S19.
[3] Halili, Festim & Kasa Halili, Merita. (2011). Analysis and comparison of web service architectural styles, and business benefits of their use.
[4] “Web API Design: The Missing Link, Best Practices for Crafting Interfaces that Developers Love”, Google Cloud
[5] https://blog.bytebytego.com/p/soap-vs-rest-vs-graphql-vs-rpc
[6] “Building Twitter's backend RPC services with Finagle by Dorothy Ordogh” https://www.youtube.com/watch?v=xXNZsD0B08k
[7] https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[8]https://restfulapi.net/soap-vs-rest-apis/