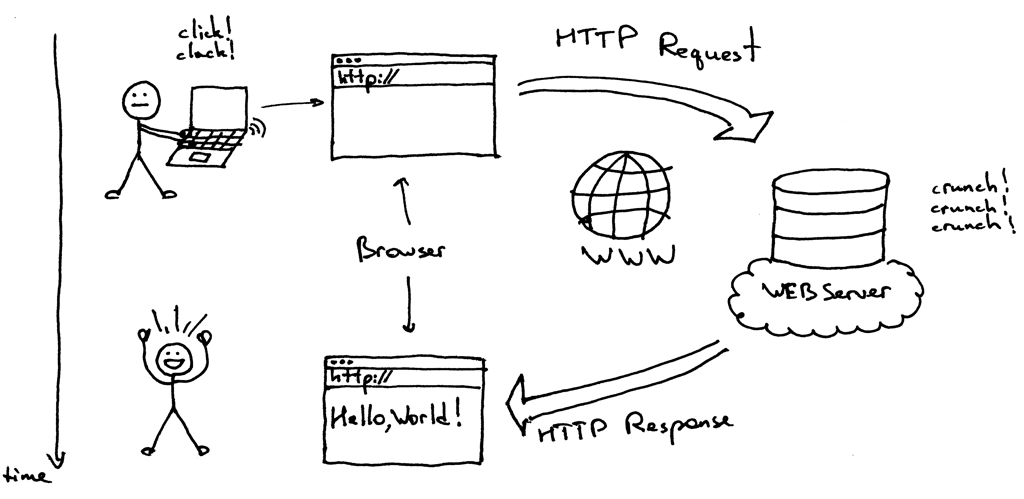
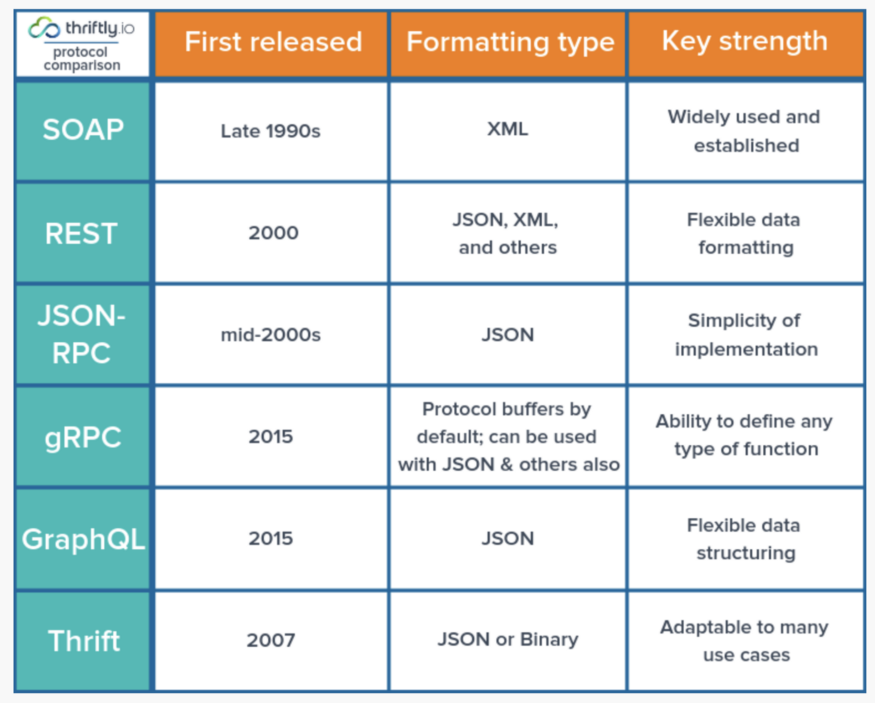


**Web Application vs Desktop Application**

| **Web Applications** | **Desktop Applications** |
| --- | --- |
| Deployment and up-gradation for a web-based application require deployment on a single set of server machines. | Deployment and any up-gradation/patch are done on individual client machines separately. |
| Web applications can be accessed from anywhere, so there is no location constraint. | As desktop are confined to a standalone machine, so they can be only accessed from the machines they are deployed in. |
| Web applications are platform-independent, they can work in different types of platforms with the only requirement of a web browser. | Desktop applications need to be developed separately for different platform machines. |
| Web applications are at higher security risks as they are inherently designed to increase accessibility. | Desktop applications, on the other hand, have better authorization and administrators have better control, hence more secure. |
| Web applications rely heavily on internet connectivity, for there operation. | Desktop applications don’t require the internet for their operations. Some applications just require internet connectivity at the time of updations. |



https://medium.com/must-know-computer-science/system-design-client-server-communication-674818ca448d



## 2.1 SOAP

SOAP is the oldest web-focused API protocol that remains in widespread use. It is an example of a Remote Procedural Call or RPC. SOAP stands for Simple Object Access Protocol. It requires a fair bit of work to make requests. You have to create XML documents to make calls, and the XML formatting that SOAP requires isn’t exactly intuitive. This not only makes call implementation difficult but also makes problems hard to debug because debugging requires parsing through long strings of complex data.

On the other hand, SOAP relies on standard protocols, especially HTTP and SMTP. That means that you can use SOAP in virtually every type of environment because these protocols are available on all operating systems.

## 2.2. REST

REST, short for Representational State Transfer, is an API protocol that was introduced in a 2000 dissertation by Roy Fielding, whose goal was to solve some of the shortcomings of SOAP.

Like SOAP, REST relies on a standard transport protocol, HTTP, to exchange information between different applications or services. However, REST is more flexible in that it supports a variety of data formats, rather than requiring XML. JSON, which is arguably easier to read and write than XML, is the format that many developers use for REST APIs. **REST APIs can also offer better performance than SOAP because they can**[**cache information**](https://restfulapi.net/caching/).

## 2.3 JSON-RPC

While REST supports RPC data structures, it’s not the only API protocol in this category. If you like JSON, you may prefer instead to use JSON-RPC, a protocol introduced in the mid-2000s.

Compared to REST and SOAP, JSON-RPC is relatively narrow in scope. It supports a small set of commands and does not offer as much flexibility as a protocol like REST with regard to exactly how you implement it. However, if you like simplicity and have a straightforward use case that falls with JSON-RPC’s scope, it can be a better solution than REST.

JSON-RPC [can be used in HTTP and WebSocket.](https://besu.hyperledger.org/en/stable/HowTo/Interact/APIs/Using-JSON-RPC-API/)

## 2.4 gRPC

gRPC is an open-source API that also falls within the category of RPC. Unlike SOAP, gRPC is much newer, having been released publicly by Google in 2015. Like REST and SOAP, gRPC uses **HTTP** as its transport layer. Unlike these other API protocols, however, gRPC allows developers to**define any kind of function calls** that they want, rather than having to choose from predefined options (like GET and PUT in the case of REST).

Another important advantage of gRPC is that when you make a call to a remote system using gRPC, the call appears to both the sender and the receiver as if it were a local call, rather than a remote one executed over the network. This simulation avoids much of the coding complexity that you’d otherwise have to contend with in order for an application to handle a remote call.

The ability of gRPC to simplify otherwise complex remote calls has helped make it popular in the context of building APIs for **microservices or Docker-based applications**, which entail massive numbers of remote calls.

## 2.5 GraphQL

In a way, GraphQL is to Facebook what gRPC is to Google: It’s an API protocol that was developed internally by Facebook in 2013. It is officially defined as a query language, also represents an effort to overcome some of the limitations or inefficiencies of REST.

One of the key differences between REST and GraphQL is that **GraphQL lets clients structure data however they want when issuing calls to a server**. This flexibility improves efficiency because it means that clients can optimize the data they request, rather than having to request and receive data in whichever prepackaged form the server makes available (which could require receiving more data than the client actually needs, or receiving it in a format that is difficult to use). With GraphQL, the clients can get exactly what they want, without having to worry about transforming the data locally after they receive it.

## 2.6 Apache Thrift

Like GraphQL, Apache Thrift was born at Facebook and functions essentially as an RPC framework. However, the design goals and target use cases for Thrift differ significantly from those of GraphQL.

Thrift’s main selling point is the ease with which it makes it possible to **modify the protocol used by a service once the service has been defined**. Combined with the fact that Thrift also supports an array of different transport methods and several different server-process implementations, this means that Thrift lends itself well to situations where you expect to need to modify or update your API architecture and implementation frequently. You might say that Thrift is great for avoiding “API architecture lock-in” because it ensures that you can easily change your API architecture whenever you need to, instead of being forced to keep the same architecture because of API inflexibility.

On the other hand, some developers might contend that the choice that Thrift gives you when it comes time to implementing the API is not ideal because it leads to less consistency than you get with other API protocols that offer only one way of doing things. If you like rigid consistency and predictability, Thrift may not be the best choice for you.