Vagner Machado

Professor Raymond Law

CSC 355

29 Apr. 2020

Application Public Interface

Modern servers rely on data exchange with clients to increase interaction and integration with third party applications. This process can be achieved through an API, or application public interface, that enables clients to request and receive data from a server. Currently, there are different schemes for implementing and defining how an API interacts with clients. Amongst those are PRC (Remote Procedure Call), REST (Representational State Transfer), and Graph QL (Query Language). While all three API implementations enable for connection and exchange of data with clients , they have fundamental differences. To cite a few of them, they differ in discoverability, complexity, and network use. This document explores the individual characteristics for each API type, their usage, strengths, and downsides.

RPC

RPC is an API implementation style that has been around for many decades and aims to model functions as means to provide a service to clients. While some argue this could be an outdated type of API implementation, RPC is still the right choice for many servers throughout the web. As the name implies, this API enables clients to execute remote calls in the server as means to interact and receive data. That is, RPC API makes existing system functions available to clients as means to interact with the API through endpoints. For example, a wine distributor’s server holds data managed by software with functions that track and analyzes product flow, best selling items, and latest additions to inventory. Benefitting from these already existing functions, this wine distributor may develop a RPC API to make this data available to liquor storeowners researching trends. This RPC API would use different endpoints to call each system function and return relevant data to the client. Hence, getting data about the latest added wines might be done through endpoint /latestProducts and best selling wines through the /bestSellers endpoint. RPC API benefits from having a medium network load and relatively easy implementation when compared to other APIs. RPC API can easily enable further interactions by creating new functions and offering new endpoints for clients to access them, which as a downside can lead to function and endpoint explosions. Additionally, making functions available to clients through endpoints makes this API highly coupled to the underlying system, creating a hurdle for the API to grow independently. Further, a predefined set of functions for a RPC API gives it low discoverability to users who, generally, cannot use the response data to discover ways to interact with the API. Aiming to decouple APIs from servers and to expand on the pre-defined set of functions and responses available in an RPC API, the REST API architecture came to exist.

Representational State Transfer

REST API offers a far more interactive way for clients to communicate with servers as its implementation stirs away from RPC’s predefined API interactions, low discoverability and high coupling to server. This dynamic API architectural style offers a greater layer of abstraction between the underlying system and the API by serving, for example, JSON responses instead of RPC’s function results. Different from RPC’s modeling of functions, REST API models resources, links between resources, and relationships amongst them. Hence, REST does not offer an endpoint but an entry point to the API, enhancing its discoverability. The root of the API serves to the client a document with metadata on available resources, how to access and interact with them, while letting the client create a custom path of navigation through the API to retrieve desired data. Enabling these complex types of interactions and abstraction, unfortunately, have some weaknesses. For example, it is hard exactly define what a REST API is and what it should support. This lack of a fixed specification causes some confusion when implementing and developing an API. Further, the REST API payload can be network heavy since document delivered by the entry point must represent the state of the whole API. Within that document, further points of interaction can be listed, which increases the potential of high network usage and chattiness. Lastly, there is not a well defined format and style for the responses received from the RESP API entry point, requiring careful thinking and planning when dealing with response object. Despite being by far the most used API style on the web, REST’s has lately sparked the need for an API that lowers network usage and chattiness, enabling interactions from areas with poor network reception. This fairly new architecture is called Graph Query Language.

Graph Query Language

This young API architecture style is still in its developing stages, yet, tech giants such as Facebook and GitHub use this model in their systems. Graph QL builds on ideas from RPC and REST while being more decoupled than the former and less then the latter. Unlike previous APIs modeling of functions and resources, Graph QL handles client queries and lets them customize the subset of data to be sent back, much like in a database system. Like REST, Graph has a well-defined schema, which allows the user to learn about the API and ways to interact with it. When compared to REST’s large document served at its entry point, Graph QL reduces the payload and network usage by sending smaller and customized data objects. Hinted in its name, Graph QL is particularly useful for graph data that contains linked relations such as Facebook’s friendships and Twitter users’ following and follower lists. The power to customize responses and to reduce network usage while maintaining a good discoverability has some shortcomings. Beyond being a relatively new API schema, it is the most complex API architecture to implement, requiring custom caching for retrieved data, and the need to build and track convoluted relationships in graph like data. Unlike REST API’s well-defined versioning standards, Graph QL’s versioning status is still in its developmental stages, which may be keeping institutions from adopting the style since versioning of the API is needed to guarantee stability over time.