Dhanesh Kesari

RPC, REST and GraphQL are 3 of the main API architecture styles. RPC's provide a more tightly coupled API, while a RESTful APIs provide more distance between the server and client. GraphQL is a mix of both RPC and REST. It provides lower latency and less calls to the API itself. Each of these has different benefits that should be considered for each use case.

An RPC API runs a function on a different computer, or space. This project is a RPC API because, although it is not being run on a different computer, we are treating it that way. We are running our functions on our server, which could be any server, not just our machine. It is incredibly easy to add functionality to an RPC, which is both good and bad. We can edit code and add new functions very easily. Network requirements are not heavy either. Depending on out application, payloads can be small, especially since we have a lot of control on what data is being sent. RPC's are great when we want to just use get and post requests. Since they are function based, we can do these things very quickly and with accuracy(we get what we want, nothing more or less). However, without proper documentation, RPC's can be hard to handle. A client may not know how to invoke your methods. There may be multiple methods with the same parameters. RPC's are great for a specific functionality, but if we want our system to me adaptable and malleable, we should look into some other API architectures.

RESTful API's model resources. Even the entry point is a model. When requesting the root of an API, we can expect a document about what the API's capabilities are. When we request from the API, we can start to get a feel of how the API is modeled. While its easier to see what is going on with a REST API when compared to a RPC API, REST API take up much more bandwidth. We may be given miscellaneous information and that can slow down our program. This extra information however, allows us to build up an API and add new functionalities even though we may not be the original developers. If our project is more future geared, and we expect it to grow, a RESTful API is probably the answer. Maybe we would use it for a forum service, enabling easier updates and changes. Now, if we needed an application that we expect to grow and would require low latency (messaging, emergency apps) we might look into a GraphQL architecture. RESTful API's also suffer from inconsistency, since the term has been used for many different architectures that may be a mix of REST and something else.

A GraphQL architecture is based off a graph structure. Instead of dealing with resources, we would create a structured query. We would pull everything we want in one request. This means that we can swiftly pull data, like RPC, and get exactly the data we need. GraphQL developers will create the API with a variety of data and the client will specify what they would like access to. This makes developing for the client much easier than would be in REST. It is also strongly-typed, so developers and clients know exactly how data is being accessed. This also means that a frontend developer or client needs to know very little about the backend to work with it. A drawback is that GraphQL is more complex than the other architecture styles mentioned. Caching can be harder as well. Versioning also has a mix of opinions in the community, where some say versioning isn't required.

A RPC architecture is good for very specific cases. These will probably be internal API's that we use to get a job done fast. Creating a consistent code base will be difficult though. With REST API's they are very good at maintaining distance between the client and server. They may, at times, give excess data. However, it is easy to be able to tell what the API does by inspection and versioning is easy as well. GraphQL is complex. It is difficult to update a GraphQL backend. However, access is easy since a client would know exactly what they have access to. There is also very low latency since we are only ever making one request. Each of these architecture styles are good in their own way. We should strive to use the correct one for the right job. We also shouldn't be extremely strict about molding our architectures to what are essentially archetypes, but be accurate what we call our own architecture styles. A mix of architectures is probably the optimal solution in most cases.