API Design

In a microservices architecture, there could be a significant amount of data exchange between services in order to fulfill a request. Consequently, APIs must be efficient to avoid creating chatty I/O. Similarly, since each microservices could be maintained by independent team, they must be designed with well-defined semantics and versioning schemes to update break.

# Public APIs vs Backend APIs

When building API, it is important to make a distinction between public and backend. Public API are those called by external clients, typically browser applications or mobile applications. For this APIs, we would often favor some form REST over HTTP. For backend API, network performance is paramount. There could be lots of traffic and services can become I/O bound. Therefore, serialization speed and payload size become important. In this case, we want to favor more efficient protocols like gRPC. Each have advantages and disadvantages.

## Interface Definition Language (IDL)

An IDL is used to define methods, parameters and returns value for an API. It can be used to generate client code and API documentation. REST over HTTP doesn’t have a standard IDL but many favor OpenAPI (i.e. Swagger). Framework like gRPC provides their own IDL and their own tools.

## Serialization

Typically, REST over HTTP utilizes text-based formats such as JSON whereas RPC would favor binary formats such as protocol buffers. Binary formats are generally faster than text-based formats but JSON has the advantages of interoperability since most languages and framework support JSON.

## Framework and Language Support

As mentioned above, HTTP and JSON is support nearly everywhere. RPC style framework such as gRPC have libraries in a limited set of languages such as C++, C#, Java and Python.

## Compatibility

If we chose to have REST over HTTP for the public API and gRPC for the backend API, we would need to have a layer where we would translate from one protocol to another. If we choose to go all the way with REST over HTTP, we could hit some performance issues but we would win in interoperability. Via performance and load testing, if the hit is acceptable, REST over HTTP could be a default choice. However, for high performance application, a framework like gRPC should make up the backbone of your backend.

# Design Considerations

## Contracts

An API is a contract between services and should only change when new functionality is added and but because of code refactoring.

## Client types

Different type of clients can have different requirement or limitation when we consider payload sizes and interactions. Consider creating different backends for each client that expose optimal interface. Remember, just because there are different backends, it doesn’t mean logic is being repeated.

## Side Effects

Any operation with side effects should be made idempotent. This will facilities safe retries and improve the resilience of the services.

## Domain Modeling

In microservices, the services don’t share the same code base or data stores because they communicate through API. Each microservices is within a domain context, a boundary. Entities sent across the wire are representation of the original.

# API Versioning

An API is a contract between services. If the API changes, there is a risk of breakage. API versioning is a way of minimizing the risks. There are a couple of ways to implement versioning.

* Have a single service support multiple version
* Use side-by-side deployment (i.e. one deployment per version and route in consequence)

However, there is a cost of maintaining multiple version so we should not create any if we don’t need them. Although semantic versioning (MAJOR.MINOR.PATCH) is an appropriate format to version services, client should only be affected by changes in MAJOR version.