Jahong Liu

System Design Basics 8: API Paradigms

API Paradigms

API Paradigms

API stands for Application Programming Interface and provides a way for clients to perform actions with servers over the network. APIs consist of a set of rules and protocols for building and interacting with software application. They define the methods and data formats that programs, typically software libraries or operating systems, should use to communicate with other software.

The three paradigms

APIs can be classified into various paradigms, including **REST**, **GraphQL**, and **gRPC**. Each paradigm possesses its own unique characteristics and excels in specific scenarios.

There are other patterns as well, such as **SOAP (Simple Object Access Protocol)** and **WebHooks**, but we will focus on the previous three paradigms since they are the most popular.

REST APIs

REST, an acronym for Representation State Transfer, refers to an API that adheres to the design principles and standard of Representational State Transfer architectural style. The concept behind REST is to utilize straightforward HTTP for communication between machines, specifically the client and the server. But there are also some the constraints associated with it.

1. **REST APIs require a client-server architecture**, where the client and server are distinct entities communicating over a network. This separation enables independent development and updates for both the client and server.

2. **REST APIs are Stateless**: Each client request to the server must include all necessary information for understanding and processing the request. The server should not retain any details about the previous client requests. This approach facilitates horizontal scaling, as the server does not need to manage or update session states or cookies.

Let’s take a closer look at state. In the context of a client sending a request to a REST API to retrieve a specific resource (GET request), the request includes all the necessary information, eliminating the need for the server to remember previous requests. Now, let’s delve into the concept of state.

Consider the example of a URL such as https: //youtue.com/videos where we want to display a list of 10 videos on a page and display more videos as the user scrolls. The constraint is that the server should not persist any state, such as how many videos have already been displayed.

Instead, data stored on the client can be sent to the server with each request. This principle is particularly relevant in pagination scenarios. When a page loads the initial 10 videos, and the user keeps scrolling down to the second page, the server does not remember that the user was already shown 10 videos.

So, the client sends this information to the server by including parameters in the GET URL. For instance, the client might send a request like:

https://youtube.com/videos?offset=0&limit=10

which fetches the first 10 videos, followed by

https://youtube.com/videos?offet=10&limit=10

which fetches the next 10 videos.

REST is an architectural paradigm, while RESTful describes the use of that paradigm. In other words, RESTful is an adjective describing web services that follow the REST constraints.

JSON

JSON (JavaScript Object Notation) is widely used data format, particularly in web operations. REST APIs not only accept data in JSON format but also respond with data encapsulated in the same format. The name `JSON` is derived from its structural similarity to JavaScript objects, featuring key-value pairs and supporting various levels of nesting. JSON serves as the preferred data format for transmitting information over the internet. It’s

important to note that both keys and values in JSON are enclosed within double quotes, indicating their type as strings.

Let's consider the following JSON object, which includes key-value pairs like firstName and lastName. Notably, this object demonstrates nesting, such as the address property containing its own properties, and the phoneNumbers array. An array within a JSON object is commonly referred to as an `array of objects`.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": ["Catherine", "Thomas", "Trevor"],
  "spouse": null
}
```

JSON is indeed one of the ways to transmit data over the internet, but it's important to note that it is not the only option available. However, JSON is often considered to have some inefficiencies.

JSON is a string format, but it specifically requires properties to beenclosed in double quotes. Using single quotes for properties would not be valid JSON.

The issues with REST APIs

While RESTful APIs are the most popular, they do have limitations that become apparent applications become more complex. One common challenge associated with RESTful APIs is the issue of either **over-fetching** or **under-fetching** data.

Over-fetching occurs when the client receives more data than necessary, leading to unnecessary consumption network resources.

Let's consider an example where we need to fetch data to display comments on a website. The required data includes the user's profile picture, username, and the comment itself. However, the /user endpoint might provide additional, unnecessary information such as country or date joined. In this case we are required to over-fetch data and subsequently filter out the irrelevant fields.

On the other hand, under-fetching occurs when endpoints are narrowly defined, resulting in the delivery of only a single field per request. This situation requires making multiple calls to fetch each required property separately. These challenges in efficiently handling data pose difficulties in complex applications.

The need for GraphQL

GraphQL emerged as an alternative API paradigm to address the limitations of REST APIs, particularly in mitigating issues related to over-fetching and under-fetching.

With GraphQL, the client gains the ability to precisely specify the required data within a single request, regardless of how the data is structured on the server. Following the client's specifications, the server takes on the responsibility of gathering all the necessary data and formatting it accordingly.

In Graph QL, there are two primary types of operations: **queries** and **mutations**. Queries are utilized to retrieve data, while mutations are employed for modifying data on the server. Notably, GraphQL operates through a single endpoint, typically an HTTP POST endpoint, where all queries are sent. The functionality of GraphQL can be effectively demonstrated using the SpaceX API.

```
{
  launchesPast(limit: 10) {
    mission_name
    launch_date_local
    launch_site {
      site_name_long
    }
    links {
      article_link
      video_link
    }
    rocket {
      rocket_name
    }
  }
}
```

This GraphQL query is designed to retrieve a set of the last 10 space launches.

```
{
  "data": {
    "launchesPast": [
      {
        "mission_name": "FalconSat",
        "launch_date_local": "2006-03-24T10:30:00+12:00",
        "launch_site": {
          "site_name_long": "Cape Canaveral Air Force Station Space Launch Complex 40"
        },
        "links": {
          "article_link": "https://www.space.com/2196-spacex-inaugural-falcon-1-rocket-lost-launch.html",
          "video_link": "https://www.youtube.com/watch?v=0a_00nJ_Y88"
        },
        "rocket": {
          "rocket_name": "Falcon 1"
        }
      },
      // ... 9 more launch objects in the same format
    ]
  }
}
```

In response, the server provides a JSON object that contains the requested data, structured in the same way as the query. The response presents the actual values rather than the property names.

To summarize, GraphQL enables us to precisely specify the required data, eliminating the need to fetch more or less data than necessary. With a singe query, we can navigate through related objects and their fields, fetching only the data we need. This approach leads to more efficient data retrieval and reduces data usage, which is particularly advantageous in mobile applications or under slower network conditions.

gRPC

gRPC stands for gRPC Remote Procedure Call making is a recursive acronym, but most people assume it stands for Google Remote Procedure Calls.

It's a framwork for executing RPCs – a method that allows a program to execute a procedure in another address space (commonly on another computer on a shared network). gRPC, just like web-sockets and HTTP/2, provides bi-directional communication, or multiplexing for multiple messages over a single TCP connection and server push.

Typically, gRPC is used for server-server communication. Performance wise, it is much faster than REST APIs since it sends data using protocol buffers, instead of JSON. Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.

gRPC also provides streaming, i.e. we can push data from the client to the server, and also from the server to the client. This might remind you of WebStockets. However gRPC is no replacement for WebSockets, it's an alternative for REST APIs.

While gRPC is built on top of HTTP/2, it does not make use of the error codes provided by HTTP. You are required to develop your own server messages.

Let's say we want to define a search request message format, where each search request has a query string, the particular page of results you are interested in, and a number of results per page. Here's the .proto file you use to define the message type.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

The protocol buffer we see above will be converted into an object in our language.

This protobuf would look like the following in Python.

```python
import protobuf

syntax = "proto3"

class SearchRequest(protobuf.Message):
    query = protobuf.Field(1, protobuf.STRING)
    page_number = protobuf.Field(2, protobuf.INT32)
    result_per_page = protobuf.Field(3, protobuf.INT32)
```

To create an actual API in gRPC, we use the following code.

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

The code defines a rather simple service called `Greeter` with two RPC methods: `SayHello` and `SayHelloAgain`.

The two lines

- `rpc SayHello (HelloRequest) returns (HelloReply) {}`
- `rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}`

define two RPC operations, as well as the request and response schemas that each will accept and return.

In this case, both methods take a `HelloRequest` message as input and return a `HelloReply` message. These are the methods that can be called using a remote machine.

The `message`'s are the schemas for the request and response payloads. In this case, the `HelloRequest` message contains a single field called `name`, which is a string. The `HelloReply` message contains a single field called `message`, which is also a string.

The messages are converted to Class definitions in most programming langauges, as shown in the Python example above. These can be used to create objects that are passed to the RPC methods.