Overview of REST vs. gRPC

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **gRPC** | **REST API** |
| Messaging Format | Protobuf (Protocol Buffers) | JSON (usually) or XML and others |
| Code Generation | Native Protoc Compiler | Third-Party Solutions Like Swagger |
| Communication | Unary Client-Request or Bidirectional/Streaming | Client-Request Only |
| Implementation Time | 45 Minutes | 10 Minutes |

**Difference between HTTP/2 and HTTP/1.1**

HTTP stands for hypertext transfer protocol & it is used in client-server communication. By using HTTP user sends the request to the server & the server sends the response to the user. There are several stages of development of HTTP but we will focus mainly on HTTP/1.1 which was created in 1997 & the new one is HTTP/2 which was created in 2015.

**HTTP/1.1:** For better understanding, let’s assume the situation when you make a request to the server for the geeksforgeeks.html page & server responds to you as a resource geeksforgeeks.html page. before sending the request and the response there is a TCP connection established between client & server. again you make a request to the server for image img.jpg & the server gives a response as an image img.jpg. the connection was not lost here after the first request because we add a keep-alive header which is the part of the request so there is an open connection between the server & client. there is a persistent connection which means several requests & responses are merged in a single connection. These are the drawbacks that lead to the creation of HTTP/2: The first problem is HTTP/1.1 transfer all the requests & responses in the plain text message form. The second one is head of line blocking in which TCP connection is blocked all other requests until the response does not receive. all the information related to the header file is repeated in every request.

**HTTP/2:** HTTP/2 was developed over the SPDY protocol. HTTP/2 works on the binary framing layer instead of textual that converts all the messages in binary format. it works on fully multiplexed that is one TCP connection is used for multiple requests. HTTP/2 uses HPACK which is used to split data from header. it compresses the header. The server sends all the other files like CSS & JS without the request of the client using the PUSH frame.

**Difference between HTTP/1.1 and HTTP/2 are:**

| **HTTP/1.1** | **HTTP/2** |
| --- | --- |
| Ithe usest works on the textual format. | It works on the binary protocol. |
| There is head of line blocking that blocks all the requests behind it until it doesn’t get its all resources. | It allows multiplexing so one TCP connection is required for multiple requests. |
| It uses requests resource Inlining for use getting multiple pages | It uses PUSH frame by server that collects all multiple pages |
| It compresses data by itself. | It uses HPACK for data compression. |

The difference between REST APIs and streaming APIs is: **Streaming APIs updates are sent to the consumer when an event happens.** **REST APIs operate in a client-server architecture**

The REST API lets you query or modify a user's account. You don't need their permission to query their account, you do need it to modify their account. They provide permission through OAuth authentication.  
The streaming API delivers tweets based on search terms or for specific users you request, along with info about the author, in real-time. You do not need the tweet author's permission. You must log into some Twitter account to use streaming, using either basic or OAuth authentication.  
Neither uses push, but streaming is a continuous net connection, so it is real-time delivery, making it functionally similar to push

Is gRPC faster than REST?

“gRPC is **roughly 7 times faster than REST when receiving data** & roughly 10 times faster than REST when sending data for this specific payload

## ****Overview of REST vs. gRPC****

To understand REST and gRPC, we need to start with APIs (application programming interfaces). APIs provide rules and definitions that allow applications to communicate and interact with each other. An API defines the types of calls and requests that one application can make to another, how to make those requests, the data formats to be used, and the conventions that clients must follow.

APIs also support the “pluggability” of applications that form a larger system because they allow two applications – even if they were written in different programming languages and running on different platforms – to communicate and interact with each other.

REST APIs and gRPC APIs refer to different architectural styles for building APIs. Here’s a brief definition of both:

* **REST (Representational State Transfer) API:** REST is the most popular architectural style for building APIs, particularly for web-based applications and microservices-based infrastructures. REST defines specific constraints that support interoperability between microservices and internet-based applications. Although a REST API can receive and return messages written in a variety of formats, the most common format used is JSON. JSON is a text-based, human-readable format that is flexible, efficient, and language/platform agnostic.
* **gRPC (Google Remote Procedure Call):** gRPC is an open-source RPC architecture designed by Google to achieve high-speed communication between microservices. gRPC allows developers to integrate services programmed in different languages. gRPC uses the Protobuf (protocol buffers) messaging format, which is a highly-packed, highly-efficient messaging format for serializing structured data. For a specific set of use-cases, a gRPC API can serve as a more efficient alternative to a REST API (more on this later).

Here’s a simple matrix that compares the basics of REST APIs and gRPC:

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **gRPC** | **REST API** |
| HTTP Protocol | HTTP 2 | HTTP 1.1 |
| Messaging Format | Protobuf (Protocol Buffers) | JSON (usually) or XML and others |
| Code Generation | Native Protoc Compiler | Third-Party Solutions Like Swagger |
| Communication | Unary Client-Request or Bidirectional/Streaming | Client-Request Only |
| Implementation Time | 45 Minutes | 10 Minutes |

### ****REST APIs****

REST (Representational State Transfer) describes a client-server organization in which back-end data is made available to clients through the JSON or XML messaging format. According to[Roy Fielding](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm), an API qualifies as “RESTful” when it meets the following constraints:

* **A uniform interface:** An API must expose specific application resources to API consumers.
* **Client-server independence:** The client and the server function independently. The client will only know the URIs that point to the application’s resources. These are usually published in the API documentation.
* **Stateless:** The server doesn’t save any data pertaining to the client request. The client saves this “state data” on its end (via a cache). **Learn more about stateful vs. stateless systems here.**
* **Cacheable:** Application resources exposed by the API need to be cacheable.
* **Layered:**The architecture is layered, which allows different components to be maintained on different servers.
* **Code-on-Demand (COD):**This is the only optional REST constraint. This allows the client to receive executable code as a response from the server. In other words, it’s the server that determines how specific things get done.

Finally, the REST API architecture generally relies on HTTP protocol, and REST APIs are the most common format for building web applications and connecting microservices. When a REST API is made publicly available as a web service, each component (or service) provided by the web service is presented to clients as a resource. Clients can access these resources via a common interface that accepts different HTTP commands like GET, POST, DELETE, and PUT.

### ****RPC APIs****

As a predecessor of REST, RPC (Remote Procedure Call) is a software architecture dating back to the 1970s. RPC allows you to invoke a function on a remote server in a particular format and receive a response in the same format. It doesn’t matter what format the server executing the request uses, and it doesn’t matter if it’s a local server or a remote server. RPC allows you to invoke a function on the server and receive the result in the same format.

The basic concept of an RPC API is similar to that of a REST API. The RPC API defines the rules of interaction and what methods a client can use to interact with it. Clients submit calls that use “arguments” to invoke these methods. However, with an RPC API, the method is found in the URL. The arguments that invoke the methods are found in the query string. To illustrate this, here’s how an RPC API request compares to a REST API request:

* **RPC:** An RPC API request might use POST /deleteResource and have a query string that says { “id”: 3 }
* **REST:** A REST API request would write this request as DELETE /resource/2.

## ****Understanding gRPC APIs****

As a variant of the RPC architecture, gRPC was created by Google to speed up data transmission between microservices and other systems that need to interact with each other. Compared to REST APIs, gRPC APIs are unique in the following ways:

1. Protobuf Instead of JSON
2. Built on HTTP 2 Instead of HTTP 1.1
3. In-Born Code Generation Instead of Using Third-Party Tools Like Swagger
4. 7 to 10 times Faster Message Transmission
5. Slower Implementation than REST

### ****When to Use REST APIs****

REST APIs are the most widely-used and popular APIs for connecting microservices-based infrastructures. Whether you are building an internal system or an open system that exposes its resources to the rest of the world, REST APIs are likely to remain the de facto choice for app integration for a very long time. Also, REST APIs are ideal when a system needs high-speed iteration and standardization of HTTP protocol.

With its universal support from third-party tools, REST APIs should be your first consideration for app integrations, microservices integrations, and web services development.

### ****When to Use gRPC APIs****

As for gRPC, most third-party tools continue to lack in-built features for gRPC compatibility. As such, gRPC is mostly relegated to building internal systems, i.e., infrastructures that are closed to external users. With that caveat in mind, gRPC APIs could be useful for the following circumstances:

* **Microservices connections:**gRPC’s low-latency and high-speed throughput communication make it particularly useful for connecting architectures that consist of lightweight microservices where the efficiency of message transmission is paramount.
* **Multi-language systems:**With its native code generation support for a wide range of development languages, gRPC is excellent when managing connections within a polyglot environment.
* **Real-time streaming:**When real-time communication is a requirement, gRPC’s ability to manage bidirectional streaming allows your system to send and receive messages in real-time without waiting for Unary client-response communication.
* **Low-power low-bandwidth networks:**gRPC’s use of serialized Protobuf messages offers light-weight messaging, greater efficiency, and speed for bandwidth-constrained, low-power networks (especially when compared to JSON). IoT would be an example of this kind of network that could benefit from gRPC APIs.

SpringBoot Request Filter:

@Component

public class SimpleFilter implements Filter {

@Override

public void destroy() {}

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterchain)

throws IOException, ServletException {

System.out.println("Remote Host:"+request.getRemoteHost());

System.out.println("Remote Address:"+request.getRemoteAddr());

filterchain.doFilter(request, response);

}

@Override

public void init(FilterConfig filterconfig) throws ServletException {}

}