ChatGPT

Representational State Transfer (REST) APIs

Historical Context

In the late 1990s and early 2000s, most web services relied on protocols like SOAP, which required verbose XML envelopes and strict message formats [1] [2]. In 2000, Roy Fielding introduced REST in his doctoral dissertation as an architectural style for distributed hypermedia systems [3] [4]. His goal was to create a *standard* way for servers to communicate over the Web, focusing on simple web concepts (URIs, HTTP methods, and representations) rather than complex messaging rules [5] [3]. REST quickly gained popularity: companies like eBay and Amazon launched RESTful APIs in the early 2000s to expose resources easily to any client [6] [5]. Today, REST is the backbone of most web and mobile applications, prized for its simplicity and alignment with HTTP.

Core Principles of REST

REST is defined by six architectural constraints (the last one optional) [4]:

- **Uniform Interface:** Clients and servers communicate through a standardized interface (URIs for resources, HTTP verbs for actions) [7] [8].
- **Statelessness:** Each request contains all the information needed to process it, so the server does not store any client session state between requests [9] [10].
- **Client-Server Separation:** The client (e.g. user interface) and server (data storage and logic) have clear, independent roles [11] [3].
- **Cacheable:** Responses must explicitly indicate whether they are cacheable, enabling clients or intermediaries to reuse them and reduce load [12] [13].
- **Layered System:** The API architecture can have multiple layers (e.g. load balancer, cache, application, database), with each layer unaware of others beyond the next step [14] [15].
- **Code on Demand (optional):** Servers may extend client functionality by sending executable code (typically scripts) that the client can run locally [16] [17].

Each of these constraints simplifies design and improves scalability and flexibility. They are typically applied together to create a truly RESTful API [4].

Uniform Interface

The **Uniform Interface** constraint standardizes how clients interact with resources. Every resource (e.g. a user, order, or image) is identified by a URI, and HTTP methods (GET, POST, PUT, DELETE, etc.) have agreed-upon meanings [7] [8]. For example, a GET request to `/users/123` retrieves user data, and a DELETE to `/posts/456` removes a post. By using a consistent interface, clients can predict how to use the API and reuse generic tools. HTTP responses typically include a status code and a body with the representation (often JSON) of the resource [7] [8].

```
GET /users/123 HTTP/1.1
Host: api.example.com

HTTP/1.1 200 OK
Content-Type: application/json

{"id": 123, "name": "Alice", "email": "alice@example.com"}
```

In this example, the client retrieves the user resource at `/users/123` using GET. The server responds with a 200 OK status and a JSON representation of the user. Note how the URI and method clearly identify the resource and action, illustrating the uniform interface. Consistent use of HTTP verbs and URIs makes the API intuitive and decouples client and server implementations [7] [8].

Statelessness

Under **Statelessness**, each request from the client must contain all information necessary for the server to understand and fulfill it [9] [18]. The server does not store any session state between requests. This means, for example, that authentication credentials or other context must be included with every request (often via headers or tokens). Because of this design, servers can treat each request independently, greatly simplifying scalability and reliability [9] [18].

```
GET /profile HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGci... (JWT token)

HTTP/1.1 200 OK
Content-Type: application/json

{"id": 1, "name": "Alice", "role": "admin"}
```

In this stateless example, the client includes a bearer token in the `Authorization` header with each request. The server uses that token to identify the user; it does not rely on any stored session. Each request is self-contained. According to AWS documentation, “the server can completely understand and fulfill the request every time” because each is independent [9]. Statelessness improves scalability (servers can easily add or remove nodes) and fault-tolerance, at the cost of requiring the client to manage more state.

Client-Server Separation

The **Client-Server** constraint enforces a clear separation of concerns: the client handles the user interface (UI) and user experience, while the server handles data storage, business logic, and security [11] [3]. This separation allows each side to evolve independently as long as they adhere to the shared interface. For example, a JavaScript frontend (client) can fetch data from a REST API (server) without needing to know the server’s database implementation. Likewise, changes in the backend (like moving to a different database) do not require changes in the client code. This modularity and decoupling is fundamental to scalability and maintainability of large systems [11] [3].

Example: A mobile app calls an API endpoint over HTTPS. The client is only concerned with sending requests and rendering responses, whereas the server focuses solely on processing requests and returning data. This clear role split is what "client-server" implies.

Cacheable

REST APIs must explicitly define whether responses are **cacheable**. If a response is cacheable, clients (or intermediate proxies) are allowed to store and reuse it for identical future requests [12] [13]. This improves performance by reducing the number of calls to the server. Servers indicate cacheability using HTTP headers such as `Cache-Control`. For example, a response with `Cache-Control: max-age=3600` tells clients they can reuse that data for the next hour without contacting the server.

```
GET /items HTTP/1.1
Host: api.example.com

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=3600

[
  {"id":1,"name":"Item1"},
  {"id":2,"name":"Item2"}
]
```

In this example, the `/items` response includes `Cache-Control: max-age=3600`, meaning the client can cache it for 3600 seconds. Caching is a built-in feature of HTTP and, when used properly, can "significantly reduce the number of requests to the server" [19]. According to AWS, caching support "improves server response time" by reusing stored data [20] [19]. By marking responses as cacheable or not, REST leverages web caching infrastructure (browsers, CDNs, proxies) to boost efficiency.

Layered System

A **Layered System** allows an API to be composed of hierarchical layers, with each layer only interacting with the adjacent ones [14] [15]. For instance, a REST API might have an HTTP load balancer in front, application servers in the middle, and a database at the back. Each layer (load balancer, cache layer, business logic layer, etc.) is independent: clients cannot tell whether they are connected directly to the server or via intermediaries [14] [15]. This layering adds modularity. Each layer can be developed or scaled independently (e.g. adding caching or security layers) without affecting the others.

- **Example layers:**
- *Presentation/API layer:* Handles incoming HTTP requests (e.g. an API gateway).
- *Business logic layer:* Processes data and applies application rules.
- *Data access layer:* Manages database/storage interactions.

By structuring into layers, REST APIs gain flexibility and scalability, since each part of the system can change internally without breaking the overall interface [14] [15].

Code on Demand (Optional)

Code on Demand is an optional constraint. It allows servers to temporarily extend client functionality by sending executable code (such as JavaScript) that the client runs. This can make clients more dynamic without requiring a full software update [16] [17]. For example, a server might respond with an HTML page that includes a `<script>` tag referencing code to run:

```
GET /dashboard HTTP/1.1
Host: app.example.com

HTTP/1.1 200 OK
Content-Type: text/html

<html>
  <head><title>Dashboard</title></head>
  <body>
    <h1>Welcome</h1>
    <script src="/scripts/chart.js"></script>
  </body>
</html>
```

Here, the server sends an HTML page that includes a `<script>` reference. The client (browser) downloads and executes `chart.js`, which was provided by the server. This illustrates code on demand: the server extended the client's capability by providing code for rendering charts. Fielding's dissertation notes that this approach "gains the separation of concerns of the client-server style" while allowing more functionality [3]. In practice, Code on Demand is less commonly used than the other constraints, but it remains part of REST's architectural definition.

REST vs GraphQL vs SOAP

REST, GraphQL, and SOAP are three different API styles. They can be contrasted along several dimensions:

Architecture

- **REST** treats **everything as a resource** accessible by a unique URL and uses standard HTTP methods to operate on them [21] [8]. For example, `GET /users/123` fetches a user. Responses are representations of resources, typically JSON. REST relies directly on HTTP features like caching and status codes [21] [8].
- **GraphQL** is a **query language** for APIs. It uses a **single endpoint** (often `/graphql`) and a schema to define what data is available [22] [23]. Clients send queries (usually via HTTP POST) specifying exactly which fields they want, and the server returns precisely that data in JSON. Unlike REST, GraphQL's design avoids hardcoded endpoints for each resource; instead, the schema and resolvers determine how data is fetched [22] [23].
- **SOAP** is a **protocol specification** that uses XML-based envelopes for requests and responses [24] [2]. SOAP messages are often sent over HTTP but also support other transport protocols. SOAP defines strict message formats, operations, and contracts (via WSDL). It focuses on standardized,

formal definitions of actions (like remote procedure calls) rather than the resource-oriented approach of REST [24] [25].

Flexibility

- **REST** is generally **flexible** in terms of data formats (JSON, XML, etc.) and loosely defined standards. Developers can design resource URIs and use HTTP methods as needed [2] [26]. This flexibility lets REST fit many use cases, from microservices to public web APIs, but it also means conventions can vary between APIs.
- **GraphQL** offers **extreme flexibility** in data retrieval. Clients can request exactly the fields they need, avoiding over- or under-fetching [27] [23]. This makes GraphQL very adaptable to changing client requirements. However, GraphQL's structure (a single endpoint and query language) can introduce complexity: caching is more involved since each query can differ, and clients must handle more logic in constructing queries [28] [29].
- **SOAP** is the **least flexible** of the three. It has a very strict, rigid structure: every message must follow the SOAP schema, and only XML is allowed [2] [30]. This rigidness limits flexibility but provides strong standardization. As one source notes, SOAP's "strict regulations lead to increased accuracy" and predictable behavior [31] [30]. In contrast to REST and GraphQL, SOAP does not allow client-driven query flexibility; operations and data formats are fixed by the service contract [30] [2].

Performance

- **REST:** Because REST uses simple resource calls, it is typically efficient for straightforward operations. REST APIs can leverage HTTP caching to reduce server load [19] [32]. However, REST can suffer from **inefficiency** when clients need related data from multiple resources. In a naive design, a client might have to make several requests (e.g. GET `/users/123` then GET `/users/123/posts`) to gather all needed data, introducing latency [33] [32]. Over-fetching or under-fetching (receiving too much or too little data) can also be an issue if endpoints return fixed data shapes [32].
- **GraphQL:** GraphQL can **improve performance** in many cases by reducing the number of network calls. Since one query can retrieve exactly the data across multiple related objects, a client often needs only a single request instead of many [33] [32]. This is especially beneficial on slow networks or mobile devices. However, GraphQL queries can be complex and may place higher computational load on the server. Also, because each query can be unique, traditional HTTP caching is less effective [28] [29]. In practice, GraphQL tends to reduce over-fetching but requires more sophisticated caching strategies (like persisted queries or client-side caching) [28] [29].
- **SOAP:** SOAP messages are typically **heavier** and more **verbose** because of XML overhead. Every request and response wraps the payload in a SOAP envelope with namespaces and headers, making them larger. This overhead can degrade performance, especially over high-latency or low-bandwidth connections [34] [35]. The DreamFactory blog notes that SOAP messages are "more verbose and complex" compared to REST, which can impact parsing speed [36]. While SOAP supports features like WS-ReliableMessaging to ensure delivery, these add additional overhead. Generally, SOAP tends to have higher latency and lower raw throughput than REST or GraphQL for the same payload size [34] [36].

Tooling and Ecosystem

- **REST:** REST has the **broadest and most mature tool support**. Virtually every programming language and web framework (Express.js, Django, Spring, etc.) has libraries for building REST APIs.

There are standardized tools for REST documentation and testing, such as OpenAPI/Swagger for schema and Postman or Curl for API calls [37] [38]. The RESTful style is universally recognized, so finding community resources and libraries (like authentication handlers, rate-limiters, etc.) is straightforward. As one source notes, "REST enjoys mature tooling (e.g., Swagger/OpenAPI...) and is natively supported by virtually all programming languages and frameworks" [37].

- **GraphQL:** GraphQL's ecosystem is **growing rapidly** but is still smaller than REST's. It has excellent tooling around its schema-centric design: tools like GraphiQL/Apollo provide interactive query explorers, and many client libraries offer generated typesafe bindings. However, because GraphQL is newer, fewer legacy systems and languages have first-class support. The community is expanding (GitHub, Shopify, and others use GraphQL), but as of now it is "not as ubiquitous as REST's" [39] [38]. Developers often rely on GraphQL-specific tools (Apollo, Relay, GraphQL Playground) rather than the general-purpose REST tools.
- **SOAP:** SOAP's tooling is mostly **enterprise-grade and specialized**. It has mature frameworks in languages like Java (.NET) that auto-generate client stubs from WSDL files. Organizations using SOAP often employ tools for WS-Security, WS-Addressing, and other extensions. However, outside of large enterprise contexts, SOAP tooling is less common. The SOAP ecosystem is well-established for its niche (banking, telecom, etc.), but it's not as lively or widely discussed among modern web developers. As one source points out, SOAP's ecosystem "may not be as dynamic as REST's or GraphQL's, but it is reliable and well-suited for certain types of applications" [38].

Use Cases

- **REST:** REST is the default choice for most web and mobile APIs today. Its simplicity and use of HTTP make it ideal for public web APIs (social media, e-commerce, etc.) and microservices. Companies like Twitter, eBay, and many others expose RESTful interfaces for general purpose data access [26] [6]. REST is especially well-suited to CRUD-style applications where resources map naturally to HTTP verbs. Its scalability and cacheability also make it a good fit for high-traffic services [26] [13].
- **GraphQL:** GraphQL is often chosen for **client-centric** or **data-driven** applications. When a frontend needs to display complex, hierarchical data (like a social media app needing user profiles, posts, comments all at once), GraphQL's one-request model shines [33] [23]. It is popular in single-page apps and mobile apps where reducing the number of network round-trips is valuable. GraphQL is also used in microservices architectures to aggregate data from multiple services into a unified schema. However, for simple CRUD apps or where caching is critical, REST may be preferred.
- **SOAP:** SOAP remains prominent in **enterprise and legacy systems** where strict contracts and advanced features are required. Industries like banking, finance, and telecommunications often use SOAP for internal services. SOAP's built-in standards (like WS-Security, transactions, and ACID reliability) make it suitable for high-security, transactional applications [40] [41]. For example, government and financial services might choose SOAP because of its robust security model (WS-Security) and formal service descriptions [40] [41]. In modern public web development, SOAP is rare, but it endures in domains that value its formality and enterprise support.

Summary: In summary, **REST** is resource-oriented and widely used for general-purpose web APIs, offering simplicity and broad tool support [21] [37]. **GraphQL** provides a flexible, query-driven approach that is well-suited to data-heavy client applications, but it requires more complex client logic and caching strategies [28] [23]. **SOAP** is a heavyweight, XML-based protocol with rigorous standards, favored in legacy and enterprise environments that need formal contracts and enhanced security [2] [40]. Each has its place depending on architectural needs: REST for most web services, GraphQL for fine-grained data fetching, and SOAP for strict enterprise integration [41] [23].

1 5 7 10 13 14 16 18 The History of REST APIs - ReadMe: Resource Library
https://readme.com/resources/the-history-of-rest-apis

2 24 What Is a SOAP API and How Does It Work? | Postman Blog
https://blog.postman.com/soap-api-definition/

3 Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)
https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

4 node.js - Can we say “If an API is following 6 constraints of REST then it’s a RESTful API”? - Stack Overflow
https://stackoverflow.com/questions/62130314/can-we-say-if-an-api-is-following-6-constraints-of-rest-then-it-s-a-restful-api

6 11 12 15 17 25 What Is a REST API? Examples, Uses & Challenges | Postman Blog
https://blog.postman.com/rest-api-examples/

8 9 20 What is REST? - AWS AppSync GraphQL
https://docs.aws.amazon.com/appsync/latest/devguide/what-is-rest.html

19 28 29 33 GraphQL vs REST: Comprehensive Comparison for 2024
https://tailcall.run/graphql/graphql-vs-rest-api-comparison/

21 22 23 26 37 39 GraphQL vs REST APIs: Key Differences, Pros & Cons Explained
https://www.getambassador.io/blog/graphql-vs-rest

27 30 32 34 38 41 REST API vs GraphQL vs SOAP | GeeksforGeeks
https://www.geeksforgeeks.org/rest-api-vs-graphql-vs-soap/

31 35 36 40 When to Use REST vs. SOAP with Examples
https://blog.dreamfactory.com/when-to-use-rest-vs-soap-with-examples