ÉCOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE (EX. INI)
OPTION: SYSTÈMES INTELLIGENTS ET DONNÉES

REST & GraphQL: A Comparative Approach

Author **E-mail**
BAAGUIGUI Ramzi *kr_baaguigui@esi.dz*

ESI - Alger, 2024

SUMMARY

1 INTRODUCTION

Web technology is all about the development of the mechanisms that allow two or more computing devices to communicate over a network. Communication between computers could never be as effective as it is without web technologies in existence. Web technology has revolutionised communication methods and has made operations far more efficient.

GraphQL and REST are two distinct approaches to designing API for exchanging data over the internet. REST enables client applications to exchange data with a server using HTTP verbs, which is the standard communication protocol of the internet. On the other hand, GraphQL is an API query language that defines specifications of how a client application should request data from a remote server.

Efficient data exchange models thus play a pivotal role in modern web development. In this report, we explore the intricacies of REST and GraphQL, their strengths, weaknesses, and real-world applications.

2 HISTORICAL ASPECT

2.1 SOAP: BEFORE REST ERA

Before REST, the API environment was a free-for-all. Developers had to deal with SOAP (Simple Object Access Protocol) to integrate APIs, which was complex and hard to debug. SOAP required hand-writing XML documents with RPC calls and following arbitrary guidelines from documentation. There was no standard for how APIs should be designed and used.

A super simple SOAP API request looks something like this:

```
1 POST http://soapl.develop.com/cgi-bin/ServerDemo.pl?class=
    Geometry HTTP/1.0
2 SOAPMethodName: http://soapl.develop.com/cgi-bin/ServerDemo.pl?
    class=Geometry#calculateArea
3 Host: soapl.develop.com (http://soapl.develop.com/)
4 Content-Type: text/xml
5 Content-Length: 494
```

Listing 2.1 – HTTP Request

```
1 <?xml version="1.0"?>
2 <SOAP:Envelope xmlns:xsi="http://www.w3.org/1999/XMLSchema/
    instance" xmlns:xsd="http://www.w3.org/1999/XMLSchema/
    instance" xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
3   <SOAP:Body>
4     <calculateArea>
5       <origin>
6         <x xsd:type="float">10</x>
7       </origin>
8       <y xsd:type="float">20</y>
9       <corner>
10        <x xsd:type="float">100</x>
11      </corner>
12      <y xsd:type="float">200</y>
13    </calculateArea>
14  </SOAP:Body>
15 </SOAP:Envelope>
```

Listing 2.2 – HTTP body content

2.2 REST: THE DILEMMA SOLVER

Back in 2000, Roy Fielding and his team had a singular goal: Establish a standard enabling seamless communication between any servers worldwide. Here's the essence of his doctoral dissertation:

> "I received feedback from over 500 developers, many of whom were esteemed engineers with extensive experience. I had to elucidate concepts ranging from abstract notions of Web interaction to intricate details of HTTP syntax. This refining process distilled my model into fundamental principles, properties, and constraints now recognized as REST."

Here are some of the principles that Roy Fielding adopted while coming with the first building blocks of REST (Representational State Transfer) model:

1. **Uniform interface:** Employing HTTP verbs (GET, PUT, POST, DELETE) consistently, utilizing URIs as resources, and ensuring each HTTP response includes status and body.
2. **Stateless:** Every request stands alone, containing sufficient context for server processing.
3. **Client-server:** Clearly delineating roles between the two systems, with one serving as the callee and the other as the caller.
4. **Cacheable:** Unless specified otherwise, clients can cache any representation, facilitated by the statelessness that renders every representation self-descriptive.

Contrary to SOAP, REST is resource-oriented, accessing nouns (URIs) rather than verbs. HTTP verbs, in turn, are utilized to interact with these resources.

While REST may seem governed by numerous regulations, these rules are universally applicable. They streamline API complexity, reducing the learning curve for developers integrating software. Roy Fielding bestowed upon the chaotic internet realm a common tongue for software communication.

2.3 GRAPHQL: CUSTOM IS BETTER

GraphQL originated as an internal project at Facebook in 2012, where it was developed to address specific challenges faced by the company's mobile applications. These

challenges primarily revolved around the need for a more efficient and flexible data-fetching mechanism. Despite being developed internally in 2012, GraphQL was not made publicly available until 2015, when Facebook released it to the developer community.

One of the primary motivations behind the creation of GraphQL was to address the complexities associated with fetching nested and varied data structures, particularly for mobile applications. Facebook's News Feed implementation on mobile devices required the retrieval of diverse sets of data with varying structures. GraphQL provided a solution by allowing clients to request precisely the data they needed, in the desired format, in a single request.

In 2018, GraphQL transitioned from being solely maintained by Facebook to being governed by the GraphQL Foundation. The GraphQL Foundation, hosted by the Linux Foundation, serves as a neutral and collaborative space for the development and advancement of GraphQL. This transition highlights the growing importance of GraphQL as a widely adopted technology beyond its origins at Facebook.

The team at Honeypot produced a documentary that delves into the history and emergence of GraphQL. This documentary provides insights into the motivations behind the development of GraphQL, its evolution, and its impact on the developer community. It serves as a valuable resource for understanding the context and significance of GraphQL in the realm of modern API development.

3 TECHNICAL ASPECT

In this chapter, we get a deep dive into the technical details of each of REST and GraphQL, taking into consideration the design philosophy of each of these API models.

3.1 REST API

3.1.1 The Philosophy Behind:

RESTful philosophy prioritizes simplicity, resource-centricity, and adaptability for scalable, reliable, and flexible APIs. It emphasizes data as URIs, independent clients/-servers, stateless communication, and hypermedia navigation, enabling gradual evolution without breaking existing services.

3.1.2 Important Concepts

3.1.2.1 Client-Server Architecture

In a RESTful system, the client and server architecture is fundamental to its design principles:

- **Client**: The client initiates requests to the server and consumes the responses. Clients can be web browsers, mobile applications, or any other software that interacts with the server.
- **Server**: The server listens for incoming requests from clients, processes them, and sends back appropriate responses. It hosts the resources that clients interact with and implements the business logic.

The separation of concerns between client and server allows for better scalability, as each can be independently scaled based on demand. Additionally, it promotes loose coupling, as changes to the server implementation do not necessarily require changes to clients and vice versa.

3.1.2.2 Request Methods

In RESTful architecture, HTTP methods (also known as verbs) are used to perform operations on resources:

- **GET**: Retrieve a resource or a collection of resources.
- **POST**: Create a new resource.
- **PUT**: Update or create a resource.
- **DELETE**: Remove a resource.
- **PATCH**: Partially update a resource.
- **HEAD**: Return headers only, not the resource representation.
- **OPTIONS**: Return supported HTTP methods for a URI.
- **TRACE**: Echo back the received request for client inspection.
- **CONNECT**: Establish a connection through a proxy.

By utilizing these HTTP methods, clients can interact with RESTful services to perform various operations on resources in a standardized and predictable manner, promoting a uniform interface and ease of use.

3.1.3 Strengths

1. Simplicity and Ease of Use: REST APIs are simple to understand and use, making them accessible to developers of varying skill levels.
2. Scalability: REST APIs are highly scalable. They operate over the HTTP protocol, which is inherently scalable and widely supported by web servers and clients.
3. Statelessness: REST APIs are stateless, meaning each request from a client to the server must contain all the information necessary to understand and fulfill that request.
4. Flexibility and Compatibility: REST APIs can support multiple data formats such as JSON, XML, and others.
5. Caching: REST APIs leverage HTTP caching mechanisms, which help improve performance and reduce server load by caching responses to requests.

3.1.4 Weaknesses

1. Over-fetching and Under-fetching: One of the weaknesses of REST APIs is over-fetching or under-fetching of data.

2. Schema Rigidity: REST APIs typically follow a fixed schema, making it difficult to evolve the API without breaking client compatibility.

3. Discoverability: Discovering available endpoints and understanding their functionalities in a REST API can be challenging, especially for large and complex APIs.

3.2 GRAPHQL API

3.2.1 The Phylosophy Behind

GraphQL embodies a philosophy of minimalism and customizability, empowering clients to request precisely the data they need without unnecessary overhead. It fosters collaboration and inclusivity by enabling transparent communication between client and server developers, promoting efficient data interactions and adaptability to evolving requirements. With its emphasis on autonomy and self-determination, GraphQL reflects a broader commitment to individual agency and flexibility in data interaction, prioritizing simplicity, efficiency, and user-centric design.

3.2.2 Important Concepts

- **Schema**: In GraphQL, the schema serves as a contract between the client and the server. It defines the types of data that can be queried and the structure of the query. The schema consists of types (e.g., object types, scalar types, enumeration types) and defines how these types relate to each other.

- **Query**: Queries in GraphQL are used by clients to request specific data from the server. A query specifies the fields and nested fields of the data that the client wants to retrieve. The server responds with JSON data that matches the shape of the query.

- **Mutation**: Mutations in GraphQL are similar to queries but are used to modify data on the server. They allow clients to perform operations such as creating, updating, or deleting data. Like queries, mutations specify the fields to be returned after the operation is completed.

- **Argument and Inputs**: Arguments and inputs are used in GraphQL to pass parameters to fields, queries, and mutations. They allow clients to customize the data they request or the operations they perform. Arguments can be of scalar types, while inputs are complex types defined in the schema.

- **Resolvers:** Resolvers in GraphQL are functions that fetch data for each field in a query or mutation. They bridge the gap between the GraphQL schema and data sources, allowing developers to customize data retrieval and manipulation logic. Resolvers are vital for handling client requests and integrating with different data storage systems.

3.2.3 Strengths

1. **Efficiency**: GraphQL enables clients to request only the data they need, reducing over-fetching and under-fetching of data. This efficiency leads to faster and more responsive applications.
2. **Strongly-Typed Schema**: GraphQL schemas are strongly typed, providing clear definitions of the data available in the API. This enables better documentation, validation, and tooling support, resulting in improved developer productivity and reduced error rates.
3. **Versionless APIs**: GraphQL APIs are inherently versionless, meaning that changes to the API schema do not necessarily require versioning. Clients can evolve independently of the server, and deprecated fields can be marked and phased out gradually without breaking existing clients.

3.2.4 Weaknesses

1. **Learning Curve**: Adopting GraphQL may require developers to learn a new query language and understand the nuances of schema design. This learning curve can be steeper for teams unfamiliar with GraphQL concepts and best practices.
2. **Complexity in Caching and Authorization**: Caching and authorization mechanisms can be more complex to implement in GraphQL compared to traditional REST APIs. Fine-grained control over caching and authorization may require additional effort and expertise.
3. **Potential for Performance Issues**: Inefficient queries or poorly optimized resolvers can result in performance issues, especially with large datasets or complex query patterns. Careful schema design and query optimization are necessary to ensure optimal performance.
4. **Potential for Security Risks**: GraphQL's flexibility can also introduce security risks if not implemented properly. For example, poorly designed queries could allow clients to request sensitive or excessive data from the server. Proper validation and sanitization of incoming requests are essential to mitigate security risks.

4 REST VS. GRAPHQL

Aspect	**REST**	**GraphQL**
Security	- Fine-grained control over security configurations at endpoint level - Risk of over-fetching or under-fetching data - Explicit access control mechanisms	- Minimizes data exposure risks - Allows clients to request only needed data in a single query - Precise query capabilities reduce unintended data exposure - Versionless nature and schema evolution capabilities reduce versioning overhead
Reliability	- Strong availability and consistency due to stateless nature - Well-defined endpoint structures - May struggle with complex error scenarios	- Consistent behavior and efficient data retrieval due to singular endpoint - Challenges in fault tolerance due to complex query structures
Performance & Efficiency	- Simplicity, efficient for straightforward tasks - May suffer from over-fetching or under-fetching data	- Precise querying enables efficient data retrieval - Poorly optimized queries can impact efficiency
Scalability	- Defined structures but may struggle with complex needs	- Allows precise data requests, adapting well to evolving requirements - Complex queries can be challenging
Adoption	- Widespread adoption and established practices - Well-understood and widely used	- Gaining popularity, especially for complex data requirements and dynamic applications - Growing community support

Tabela 4.1 – Comparison between REST and GraphQL

5 REST VS. GRAPHQL: HOW TO MAKE THE RIGHT CHOICE

Here are some guidelines based on which we decide to choose between GraphQL and REST.

- **Data Requirements** Evaluate the complexity of your data needs. If your application has simple data requirements and follows a standard CRUD (Create, Read, Update, Delete) pattern, REST might suffice. However, for complex data structures or precise data fetching needs, GraphQL could be more suitable.
- **Client Requirements** Consider the needs of your clients or consumers. If they require flexibility in data fetching and want to avoid over-fetching or under-fetching of data, GraphQL's precise querying capabilities might be preferred. REST might be sufficient if clients are comfortable with predefined endpoints and data structures.
- **Developer Expertise** Assess the familiarity and expertise of your development team. If your team has extensive experience with RESTful APIs and is comfortable with its constraints and best practices, sticking with REST might be pragmatic. However, if your team is open to learning new technologies and exploring more flexible data querying options, GraphQL could be an exciting choice.

6 CONCLUSOIN

In conclusion, our comparative study between REST and GraphQL has shed light on the distinct advantages and disadvantages of each API architecture. Through a thorough examination of factors such as data requirements, client preferences, developer expertise, and scalability considerations, we have gained valuable insights into the strengths and limitations of both REST and GraphQL.

While REST offers simplicity, widespread adoption, and well-established practices, it may struggle with complex data fetching needs and evolving client requirements. On the other hand, GraphQL's flexible querying capabilities, precise data retrieval, and versionless schema evolution present compelling advantages, particularly for applications with dynamic data requirements and varied client needs. However, GraphQL's learning curve and complexities in query optimization may pose challenges for developers.

Our study has emphasized the importance of carefully evaluating project requirements and considering the trade-offs between simplicity, flexibility, and performance when choosing between REST and GraphQL. By aligning API architecture decisions with the specific needs and goals of the application, developers can make informed choices that optimize data retrieval efficiency, enhance developer productivity, and improve overall system reliability.

In summary, while REST and GraphQL offer distinct approaches to building APIs, there is no one-size-fits-all solution. Rather, the selection of the right API architecture should be guided by a comprehensive understanding of the project's requirements, an appreciation of the strengths and weaknesses of each approach, and a commitment to delivering scalable, efficient, and user-centric solutions.