**Optimizing Microservices with GraphQL vs REST:**

**A Comparative Analysis**

**Introduction**

Microservices architecture revolutionized software design by enabling the creation of scalable applications through independently deployable services. This approach promotes agility in development, allowing teams to work on different services simultaneously without dependencies. It enhances fault tolerance, as failures in one service are less likely to impact the entire system. Moreover, microservices support the use of diverse technologies and programming languages for different components, enabling teams to choose the best tools for specific tasks. Ensuring efficient communication between these services is crucial for maintaining system responsiveness, reliability, and seamless user experiences.

This project explores the implementation of a microservices architecture, emphasizing service design through both GraphQL and REST technologies. We perform a comparative analysis of their performance to evaluate their effectiveness in modern web application frameworks.

|  |  |  |
| --- | --- | --- |
| Feature | REST | GraphQL |
| Definition | REST (Representational State Transfer) is an architectural style with rules and constraints for creating web services using HTTP methods. | GraphQL is a query language and specification for building APIs, typically using a single endpoint (/graphql). |
| Request Methods | Uses HTTP methods like GET, POST, PUT, and DELETE for different actions. | Uses a single endpoint for all requests, regardless of the type of action. |
| Data Fetching | Fetches all records or data defined by the server's response. | Allows the client to specify precisely what data it needs, optimizing payload size. |
| Documentation | Requires additional effort to create comprehensive documentation for the API. | Offers a self-documenting schema, making API structure easier to understand and query. |

**Application Overview**

The purpose of this application is to provide an easy-to-use interface for retrieving movie data from a database, either via a RESTful API or a GraphQL API. Users can query for movie information based on two key parameters:

1. **Number of Years (num\_years):** Fetch movies from the past N years.
2. **Specific Year (year):** Fetch movies from a specific year.

The application supports two types of endpoints for accessing the movie data:

* **REST API:** Offers a simple way to retrieve movie data by year or number of years. Users can send a GET request to retrieve the movie list for the specified parameters.
* **GraphQL API:** Provides a more flexible query interface, allowing users to specify exactly what movie data they want. Users can query for movies based on the number of years or a specific year, and retrieve selected fields such as title, genre, director, actors, plot, and more.

The **key features of the application** include:

* Fetching movie data based on the number of years or a specific year.
* Ability to return movie details such as title, year, genre, director, actors, plot, awards, and poster image.
* Two APIs (REST and GraphQL) that offer different approaches to data retrieval, with REST offering simplicity and GraphQL offering flexibility in specifying data needs.

This application is ideal for accessing and managing movie data in a way that allows for efficient querying based on temporal criteria (such as year or range of years). It serves as a flexible solution for both simple and complex data access needs.

**Design Considerations**

The choice between REST and GraphQL depends on the specific needs of our project. REST is suitable for straightforward, cacheable communication protocols and is often preferred for public-facing APIs. GraphQL on the other hand, is ideal for complex applications with rapidly changing front-end requirements, offering greater flexibility and efficiency in data retrieval.

Following design choices were considered while implementing the application using both REST and GraphQL frameworks.

1. **Architecture**

**REST:**

Operates through multiple endpoints, with each endpoint exposing a specific resource. Clients interact with these endpoints using various HTTP methods (GET, POST, PUT, PATCH, DELETE) to perform different operations. The decentralized nature of endpoints can result in a more complex API structure, often requiring additional management for versioning

**GraphQL:**

Adopts a single-endpoint approach, typically /graphql, to handle all queries and mutations. Clients send GraphQL queries to this endpoint, specifying exactly what data they need. Simplifies the API structure and reduces the overhead associated with endpoint proliferation and version control.

1. **Efficiency**

**REST:**

Fixed data structures returned from each endpoint can lead to inefficiencies. *Over-fetching:* Receiving more data than necessary, *Under-fetching:* Requiring multiple requests to gather all required data. Such inefficiencies can impact applications with limited bandwidth or real-time data requirements.

**GraphQL:**

Enables clients to request only the specific data they need in a single query. Avoids the pitfalls of over-fetching and under-fetching, optimizing network usage particularly beneficial for mobile and low-bandwidth scenarios.

1. **Schema design**

**REST:**

Lacks a built-in schema or type system. Data structures are defined on the server, requiring clients to adapt to these predefined formats and relies heavily on manual documentation for API structure and usage

**GraphQL:**

Employs a strongly typed schema written in GraphQL Schema Definition Language (SDL). This schema acts as a contract between the client and server, ensuring data consistency and facilitating tooling and automation such as automatic code generation and API documentation

1. **Error handling**

**REST:**

Uses standard HTTP status codes to indicate errors. Each status code has a specific meaning, such as 200 for success, 400 for client errors, and 500 for server errors

**GraphQL:**

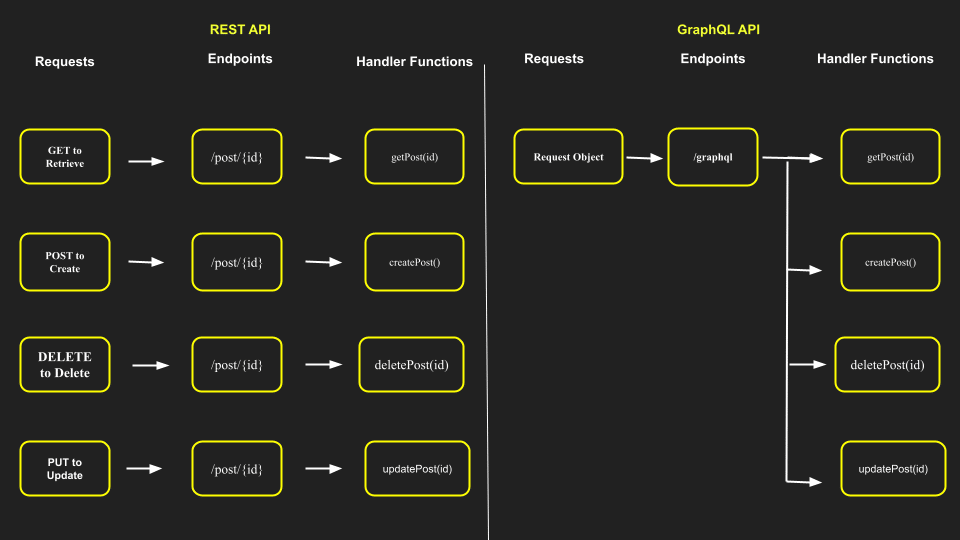
Delivers errors in the response payload alongside the correct data. This approach provides a more nuanced error handling mechanism, where the "errors" entry in the response body details the specific errors encountered

**System Architecture**

Below Architecture diagrams depict the approach taken towards designing the service interaction between various system components both in the case of REST and GraphQL.

|  |  |
| --- | --- |
|  |  |
| REST | GraphQL |

The architecture of a web application using REST is characterized by multiple endpoints, fixed data structures and the potential for over-fetching or under-fetching. In contrast, GraphQL uses a single endpoint, allows for flexible and precise data retrieval and supports real-time updates and continuous schema evolution without versioning. These differences make GraphQL more suitable for complex applications with rapidly changing front-end requirements, while REST remains a good choice for simpler, more structured APIs.



Following are the key design approach differences when building a web application using REST vs GraphQL

1. **Endpoint Structure**: REST uses multiple endpoints, while GraphQL uses a single endpoint.
2. **Data Retrieval**: REST returns fixed data structures, while GraphQL allows clients to specify exact data needs.
3. **Schema and Type System**: REST does not enforce a schema, while GraphQL uses a strongly typed schema.
4. **Versioning**: REST requires versioning, while GraphQL does not.
5. **Network Requests**: REST can lead to multiple network requests, while GraphQL reduces network round trips.
6. **Error Handling**: REST uses HTTP status codes, while GraphQL includes errors in the response payload.
7. **Real-Time Updates**: REST does not support real-time updates out of the box, while GraphQL does through subscriptions.
8. **Flexibility and Customization**: GraphQL offers more flexibility in data retrieval compared to REST.

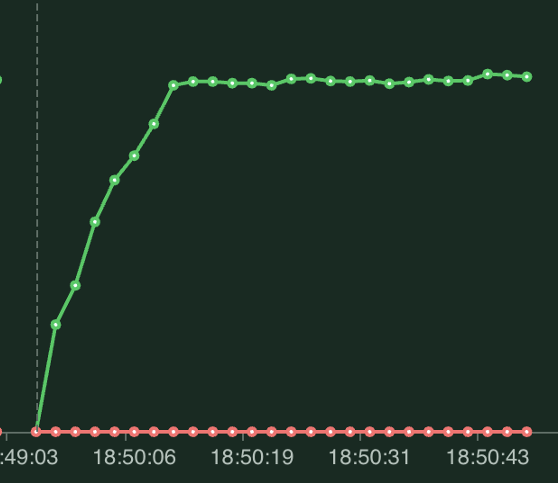
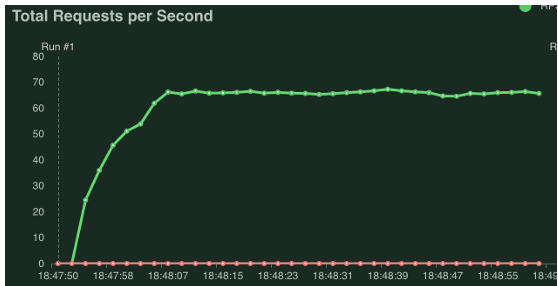
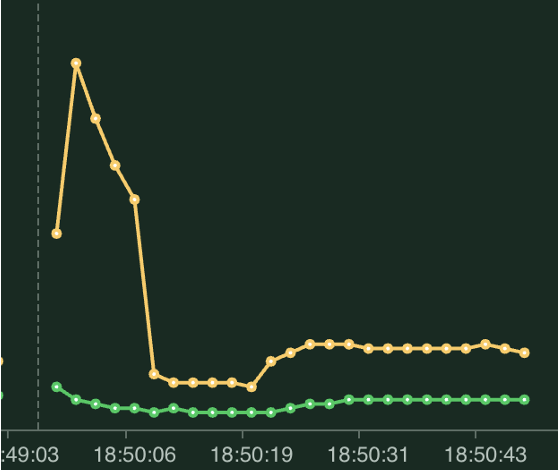
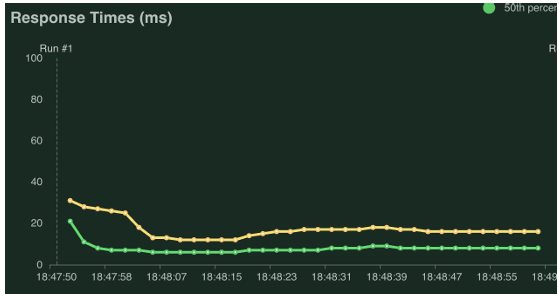
**Experimentation & Performance testing**

We undertook load testing of the application by simulating 100 concurrent users and a Spawn rate of 20 secs. For this purpose, we used Locust. Locust is an open-source load testing tool designed to evaluate the performance and scalability of various systems, including websites, applications, and other networked services. Locust allows users to define test scenarios in plain Python code, making it easy to simulate real-user behavior. Users can create scripts that mimic the actions of real users, such as navigating through a website, logging in, and performing various tasks.

Following are the load testing results as recorded during experimentation with the application.

**REST test statistics**

****



**GraphQL test statistics**

****

**A graph with lines and numbers

Description automatically generatedA graph showing a number of numbers

Description automatically generated with medium confidence**

**Conclusion & Analysis**

In a test scenario involving 100 users with a spawn rate of 20, GraphQL demonstrates better performance in terms of data fetching efficiency and reduced network requests.

GraphQL performs better in terms of data fetching efficiency. With GraphQL, clients can specify exactly the data they need, reducing over-fetching and under-fetching issues. This results in fewer network requests and less redundant data being transferred. For example, if each user needs to fetch a user profile along with their posts, GraphQL can handle this with a single query, whereas REST would require multiple requests.

**GraphQL**

* Fewer network requests
* Reduced data fetching time
* More efficient handling of complex data relationships
* Potential drawbacks in caching efficiency

**REST**

* Higher number of network requests
* Increased latency due to multiple round trips
* Better caching efficiency, though this may be less significant at high spawn rates
* Simpler error handling using HTTP status codes