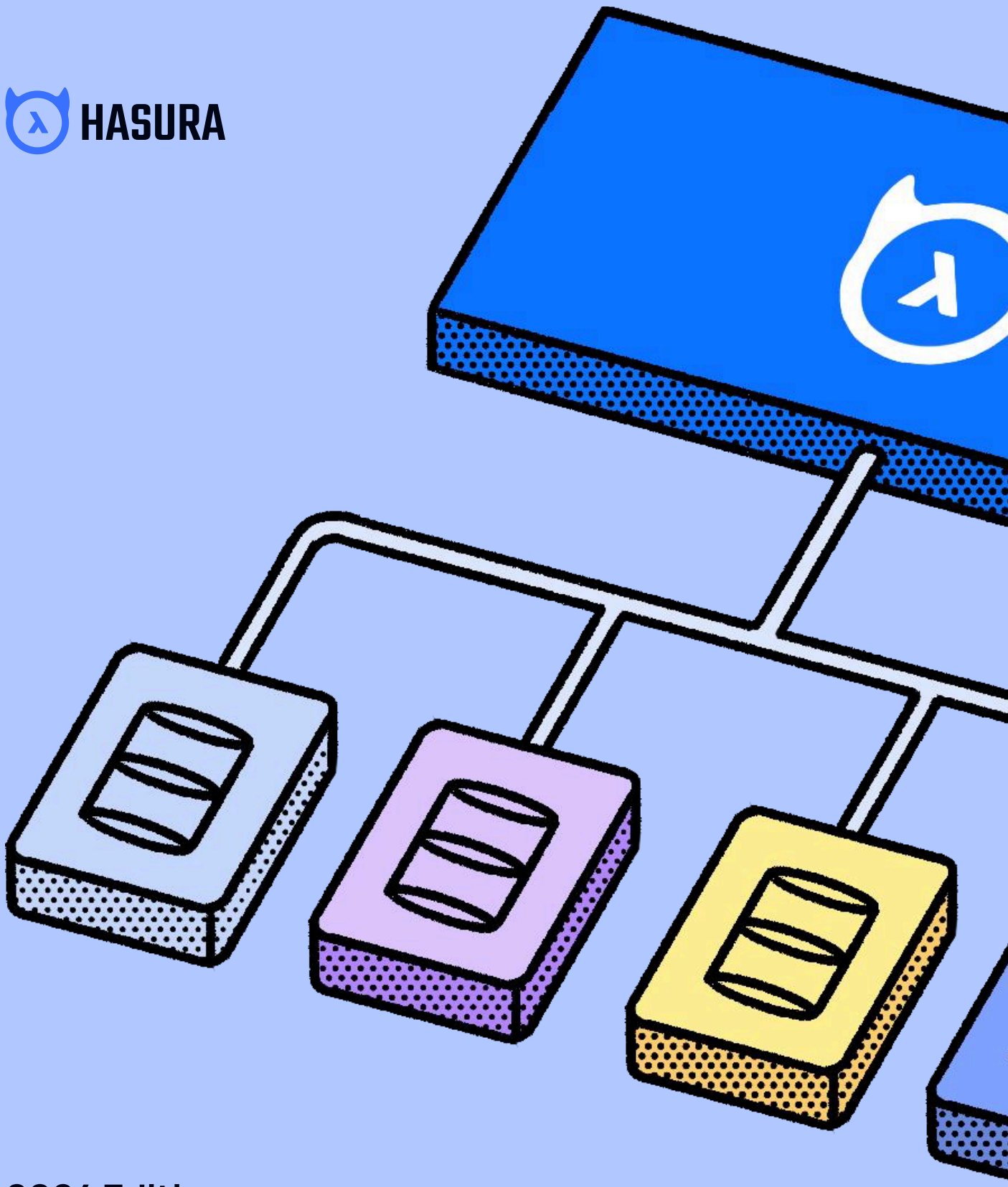
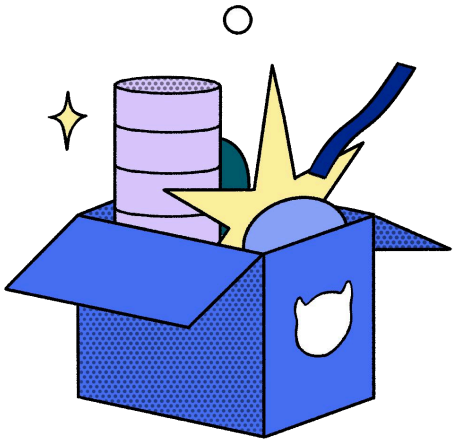
HASURA



2024 Edition

The GraphQL Federation Handbook



Introduction

This is a handbook on GraphQL federation designed for developers and architects in API platform teams. Whether you are a seasoned professional working in a large organization or just beginning to explore the realm of GraphQL federation, this guide will help you navigate the various terminologies, the tooling landscape, and the challenges and pain points in your federation adoption journey. This guide also aims to go beyond the basics of GraphQL federation to address API management and related concerns pre- and post-federation adoption.

How to contribute

We've created a GitHub repository to promote discussions in the GraphQL ecosystem and to allow everyone to contribute to this report. Please use this repository to create issues, discussions, and to submit PR contributions.

Contents

What is GraphQL federation?

Federation in GraphQL is the ability to declaratively compose multiple GraphQL schemas into one unified graph that captures the relationship between types across source schemas. A single GraphQL endpoint abstracts away the underlying sources of data and APIs.
A federated GraphQL API brings about the benefits of the monolith for consumers and microservices for producers of the API.

Typically in organizations, APIs are built by semi-independent teams with some overlapping functions. The documentation for the APIs may exist, but something else is needed to combine or connect these APIs. They are available independently for consumption. As use cases grow on the consumption side, the need to connect these APIs has risen. GraphQL's schema and type system and query capabilities make it a great choice to connect these APIs.

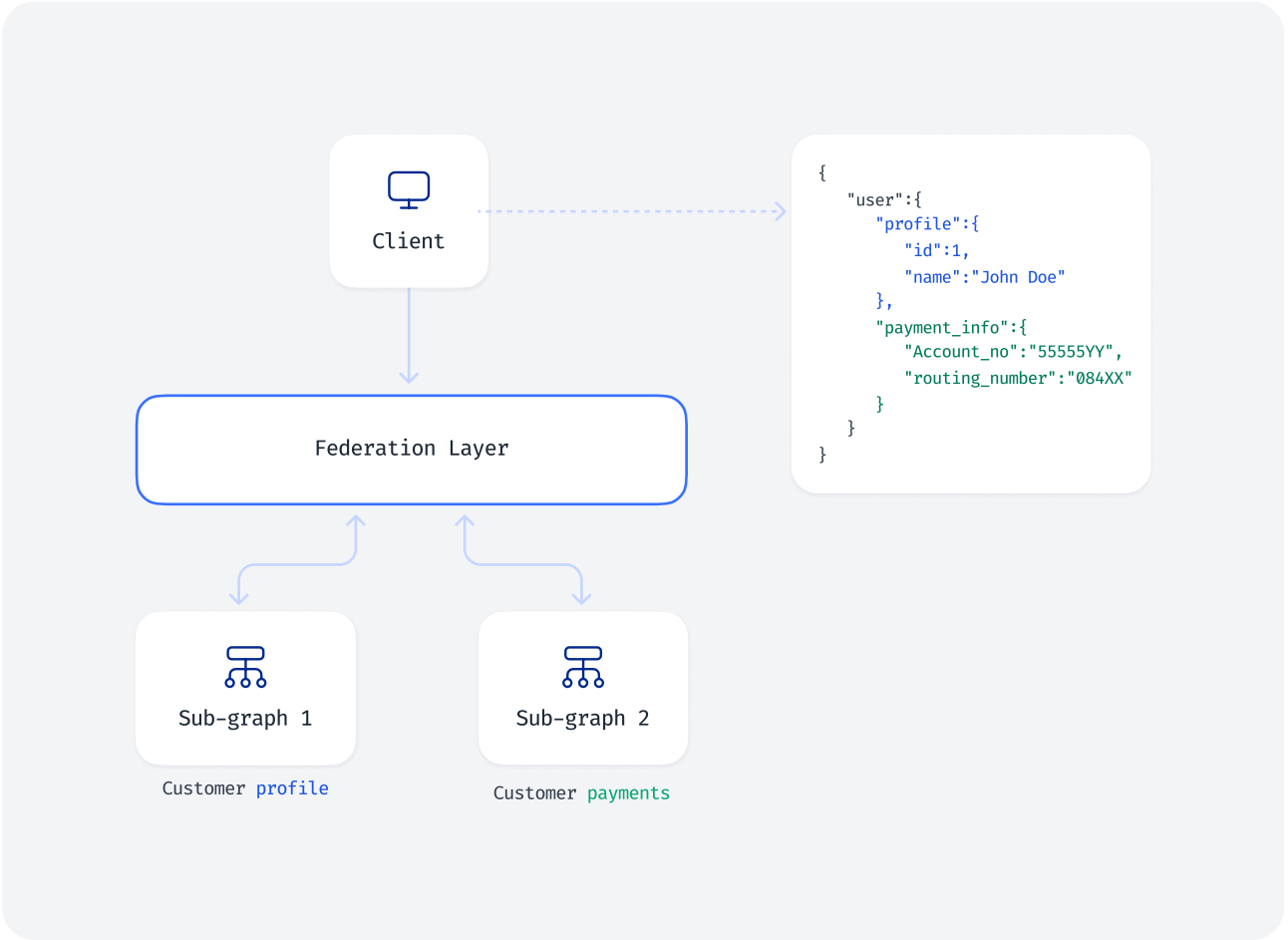
Let us take a look at a quick example:

```
query {
  authors {   → data comes from authors subgraph
    id
    name
    articles {  → data comes from articles subgraph
      id
      title
    }
  }
}
```

The client makes the above GraphQL query to fetch authors and articles. The client doesn't need to know whether the data comes from a single source or multiple sources. The author's API could have been built by one team and another could have built the articles API.
The independent APIs are referred to as subgraphs. GraphQL federation allows for this type of querying across different APIs and sources.

Architecture

An example architecture of a simple GraphQL federation use case, where there are two independent GraphQL subgraphs (i.e. endpoints), composed together via a federated gateway in the middle.



GraphQL federation evolved from schema stitching, which is a native unification of multiple schemas.

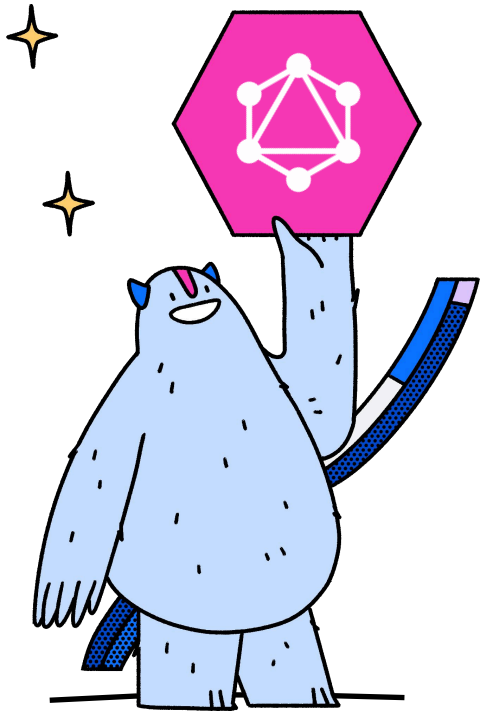
GraphQL federation vs. schema stitching:

- Schema stitching is done at the schema level whereas GraphQL federation is done at the API / service level.
- Stitching of GraphQL schemas could be done at the code level using libraries or programmatically using scripts. Federation in GraphQL requires a gateway layer in the middle to compose the federated schema and execute the queries across services.
- GraphQL federation offers schema composition, easier type conflict resolutions, and namespacing, among other things. Schema stitching typically requires schema modifications when conflicts arise.
- In a federated architecture, teams in an organization can independently operate GraphQL APIs and only the federated gateway needs to understand the dependencies between various services. Schemas stitching requires close collaboration when there are type conflicts.
- Common solutions: Schema stitching can be done using graphql-tools in JS, graphql-braid in Java, etc. Popular federation solutions are Hasura, Apollo Router, GraphQL Mesh, etc.

	Schema stitching	GraphQL federation
Operates at	Schema level	API or services level
How does it work?	Stitch schemas programmatically with code or using libraries	Federate APIs with a gateway layer to compose
Conflict resolutions	Requires modifications in the schema	Offers schema composition, easier type conflict resolutions, namespacing
Team collaboration	Requires close coordination to merge and resolve conflicts	Teams can independently operate and gateways can understand and navigate dependencies
Popular solutions	graphql-tools (JS), graphql-braid (Java)	Hasura, Apollo Router, GraphQL Mesh

GraphQL federation vs. data federation:

GraphQL federation is a pattern within the GraphQL ecosystem that enables the composition of multiple GraphQL services into a unified schema. This is scoped specifically to GraphQL APIs and the composition of GraphQL schemas. Data federation is a broader pattern that involves creating a unified API from disparate data sources. The scope and impact of data federation go beyond just composing GraphQL APIs and extend to composing data from databases, REST/ GraphQL/gRPC APIs.

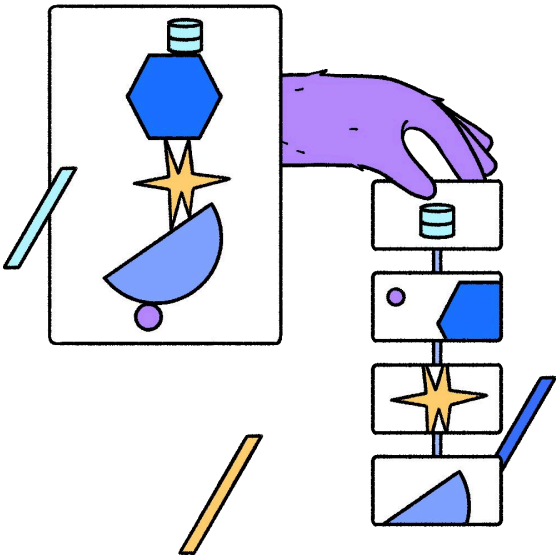


Official GraphQL federation specification

GraphQL has an official specification backed by the GraphQL Foundation. Federation in GraphQL currently doesn't have a formalized specification. There are implementations from Apollo with their Apollo Federation v1 and v2 specifications, and there was a brief community-contributed vendor-neutral attempt with GraphQL Fusion.

There's an active effort in the GraphQL Foundation to standardize the specification for federating GraphQL APIs. Learn more about the Composite Schemas Spec.

Hasura is actively involved in the foundation working group to standardize the specification. Currently, Hasura works as a federation layer for any subgraph source. Hasura also supports Apollo Federation spec in the subgraph to integrate with Apollo Router.



Why do you need GraphQL federation?

GraphQL federation is an essential tool for modern API management, providing a unified approach to managing and querying multiple GraphQL services. The following are GraphQL federation core benefits:

- **Unified API endpoint:** combine multiple GraphQL schemas into a single unified endpoint. This simplifies the client-side experience, as clients can access data from multiple sources through a single API, eliminating the need for multiple requests and reducing the complexity of managing multiple endpoints.
- **Flexibility in data sources:** Integrate with other types of data sources, such as REST APIs, databases, and microservices. This flexibility allows you to create an abstracted data layer that meets the diverse needs of modern apps.
- **Team collaboration:** Promote better collaboration among development teams. Each team can independently develop, deploy, and maintain its own GraphQL service (subgraph). The federated gateway then stitches these subgraphs into a cohesive whole. This separation of concerns allows teams to work autonomously without stepping on each other's toes, fostering collaboration and efficiency. Additionally, it ensures that changes in one part of the system do not inadvertently affect other parts, promoting stability and robustness in the overall API ecosystem.

There are other benefits like the enhanced query capabilities on the client side, where, the client only needs to make one request instead of multiple HTTP roundtrips.

Challenges and pain points

Operating a federated GraphQL API goes beyond connecting and composing a few GraphQL endpoints. When multiple teams contribute to the same API without working on the same codebase, several challenges and pain points arise.

Let's look at common challenges:

Conflict resolution

A common blocker in the unification of GraphQL schemas – type conflicts. Different subgraphs may have overlapping or conflicting fields and types, which could lead to inconsistencies or unexpected behavior when merging schemas. A simpler way to reduce impact is to follow good schema design and naming conventions.

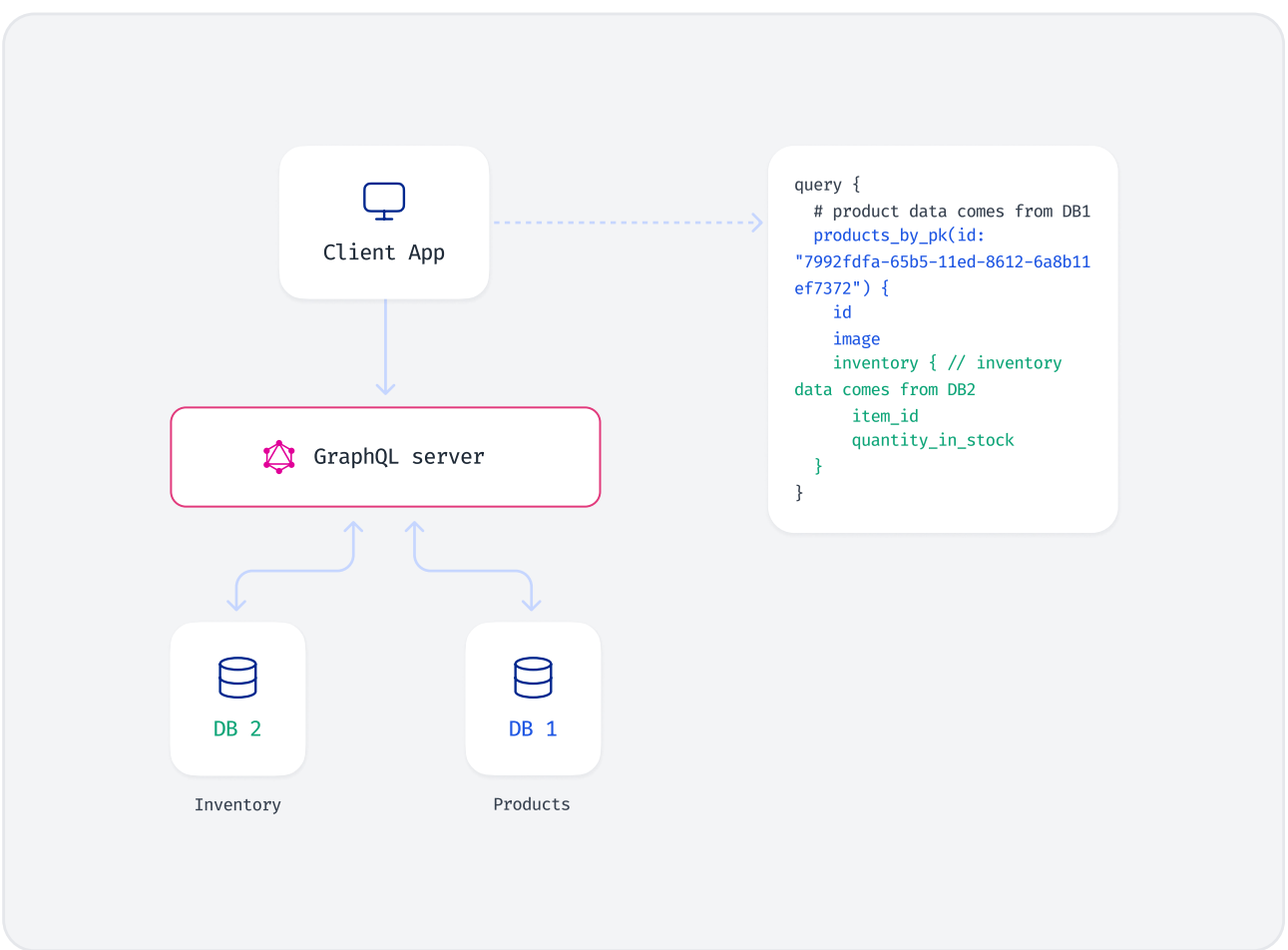
Some common schema governance rules to solve type conflicts:

- Namespacing subgraphs
- Standardized naming conventions
- Deprecating conflicting fields
- Transforming field names (add prefix or suffix) declaratively

Performance

Network latency between client and server will almost always be higher than network latency between co-located API servers and data sources. It's better to do fewer roundtrips from the client, and this is where GraphQL federation helps with the ability to query multiple sources (related or unrelated) in the same HTTP request.

Unfortunately, this means that the client-side batching/roundtrip problem is delegated to the server. Whether the client uses GraphQL or REST to solve the federation problem, the load on upstream services will be the same. The upstream services could be databases or more REST APIs underneath.



> How do you reduce the load on upstream services like the database or APIs?

Fundamentally you have to optimize for the following concerns:

- The number of queries sent to the upstream services (N+1 query problem) by the federated gateway
- The subgraph latency should be low
- Query response caching at the federated gateway layer

Let us address the concerns:

Concern 1:
The number of queries to the upstream services (N+1 query problem).

In the case of GraphQL, the number of queries is a fundamental problem, even for a single data source, but this is largely mitigated by using a DataLoader pattern/batching. In the case of REST APIs, ORMs will have similar problems and the best you can do is batch the queries to reduce the load.

Essentially, you need a good routing solution and an efficient federated server to batch queries across data sources.

Concern 2:
The subgraph latency should be low.

This is related to the first concern, especially if the database is the upstream service. Batching solves the problem of sending too many queries. But you're still sending multiple queries.

At the outset, you want to reduce the number of unnecessary hits to the database as much as possible. As the number of requests to the database increases, the number of active connections will increase and you will have to start managing connection pooling.

If you're using GraphQL to make APIs as flexible and composable as possible, you need an efficient query planner. Unlike REST APIs, where you need to optimize every query individually, GraphQL lets you "compile" them, irrespective of the depth of the query.

LinkedIn built its query planner using Apache Calcite. Goldman built the Legend project from the ground up. Read more about how to optimize GraphQL queries for high performance.

Concern 3:
Query response caching at the federated gateway layer.

One of the ways to improve performance is to cache response data before it hits upstream services. A federated gateway solution should be able to cache at the gateway level instead of leaving the logic to the subgraphs.

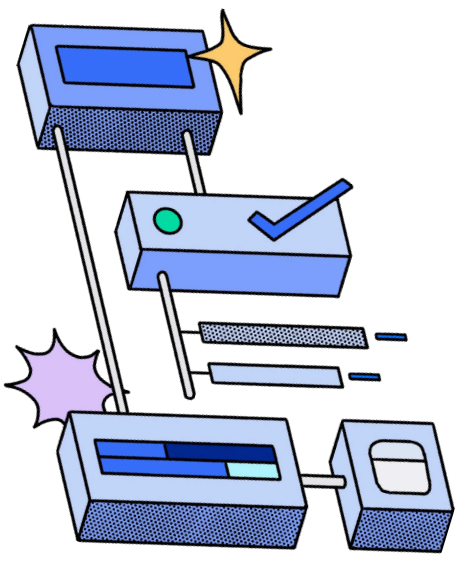
Hasura can cache any remote GraphQL API. Read more about how Hasura helps with caching here.

Security

Subgraphs may have authentication logic and in some cases independent authentication methods. A federated GraphQL API should ideally manage authentication logic at the gateway level.

A federated GraphQL API should ideally enforce authorization rules when there are shared and related entities in the graph. Additionally, producers of the subgraph should be able to forward HTTP request headers and delegate authorization logic to the subgraphs in cases with existing logic.

Security attribute	Description
Restrict direct access to subgraphs	The federated GraphQL API should be the entry point for API consumption. Directly communicating with the subgraphs should be avoided since it can bypass other security configurations, and will be difficult to analyze or trace requests.
Query depth limit and node limit	Configure GraphQL query depth limit and the number of top-level nodes that can be queried.
Rate limiting	Implement rate limiting based on user information / IP address.
Limit queries by adding an allowed list of queries	Limit the exact type of queries that can made against the federated API.
Disable introspection	Disable GraphQL introspection in production to avoid exposing
Authentication	Enforce authentication via JWT or webhooks.
Authorization	Enforce permission rules for the unified GraphQL API.



Quality criteria for subgraphs

Ensuring high-quality subgraphs is crucial for the success of a federated GraphQL architecture. The following quality criteria should be met to maintain the integrity and performance of the supergraph:

1. Standardization attributes

- **Queryable models vs commands:** Models should be collections of data that can be queried using standardized operations. Commands should map to specific business logic.
- **Conventions for query operations:** Establish standardized syntax and conventions for common query operations, including joins, filtering, pagination, sorting, and aggregations.

2. Composability attributes

- **Joining data:** The ability to join related data, similar to foreign key relationships in databases.
- **Nested filtering:** Enable filtering of parent entities based on the properties of their child entities.
- **Nested sorting:** Allow sorting of parent entities based on the properties of their child entities.
- **Nested pagination:** Provide paginated lists of parents, along with paginated and sorted lists of children for each parent.
- **Nested aggregation:** Facilitate aggregation of child entities in the context of their parents.

3. CI/CD integration

- **Seamless CI/CD process:** Ensure that subgraph updates integrate smoothly into the continuous integration and delivery pipeline, minimizing disruption and maintaining synchronization with domain changes.
- **Preventing Breaking Changes:** Implement robust change management processes to avoid breaking changes in API contracts and ensure up-to-date documentation.

4. Performance considerations

- **Low latency and high throughput:** Subgraphs must be optimized to maintain or improve performance compared to direct access to underlying domains, focusing on effectively reducing latency and managing payload size and concurrency.
- **Efficient query resolution:** Utilize efficient query planning and execution strategies to minimize the number of queries and optimize data retrieval.

5. Security and governance

- **Authentication and authorization:** Centralize security policies at the federated gateway to manage authentication and authorization consistently across all subgraphs.
- **Restrict direct access:** Ensure that subgraphs are accessed only through the federated gateway to maintain security and traceability.

6. Self-Serve connectivity

- **Ease of integration:** Domain owners should be able to connect their domains to the supergraph with minimal overhead, promoting self-serve access and iteration.
- **Comprehensive documentation:** Provide detailed documentation to help API consumers discover and utilize the API efficiently.

By adhering to these quality criteria you can ensure that each subgraph contributes effectively to a robust, scalable, and secure federated GraphQL architecture, facilitating seamless data access and API integration.

You can read more about the quality of subgraphs and how they affect the federated graph in this Supergraph Architecture Guide.

Hasura in your federation strategy

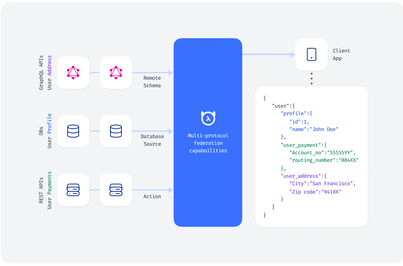
For architects and developers on platform teams, Hasura offers a robust and user-friendly solution for managing APIs across multiple data sources. Its ease of use, automation, and comprehensive feature set provides a significant edge over custom-built solutions that often demand substantial time and effort for design and implementation.

With Hasura, platform teams can seamlessly establish a unified GraphQL API that connects to diverse data sources, empowering developers to focus on building innovative applications that leverage this data without the complexities of API management. This approach allows architects to craft efficient, scalable platforms that accelerate development while simplifying data integration.

Here's how you can leverage Hasura in common federation patterns:

Multi-protocol data federation

Hasura works with data from multiple protocols or sources like databases from differing providers, REST, and GraphQL APIs. Hasura enables federating data across these disparate sources by creating relationships between types agnostic of their source.



When to use this architecture

Technical standpoint

- Your team has access to all the sources (databases, REST, and GraphQL APIs) and you want to federate these sources into a unified graph.
- High-performance joins across databases are required.

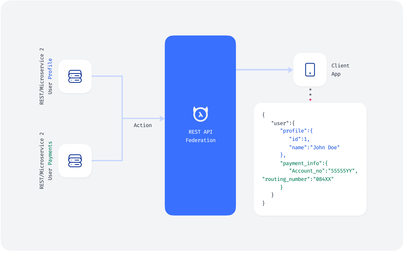
Ownership standpoint

- Your team has access to all the sources (databases, REST, and GraphQL APIs) and you want to federate these sources into a unified graph.

Hasura as a GraphQL gateway to REST APIs/microservices

Hasura allows you to expose existing REST APIs behind a single GraphQL API by declaratively adding these REST APIs as Actions in Hasura without any middleware by leveraging request/response transforms. Additionally, you can also create relationships across types from different REST APIs.

This architecture pattern allows you the flexibility to rapidly evolve this graph by connecting Hasura to the underlying databases.



When to use this architecture

Technical standpoint

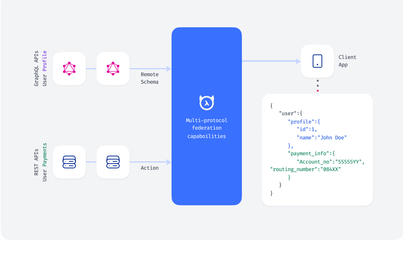
- You want to expose a GraphQL API to consumers while leveraging existing REST APIs and/or microservices.
- To accelerate development on this API by connecting Hasura to any underlying databases, the database(s) must be one of those supported by Hasura like PostgreSQL, SQL Server, MySQL, Oracle, etc.
- You want to query data from multiple REST APIs in a single request from a client application.

Ownership standpoint

- Your team may or may not own or manage the underlying REST APIs but are allowed to consume or expose the APIs.

Hasura as a federated gateway

Hasura can act as a centralized federated gateway to multiple multi-protocol (REST and GraphQL) subgraphs owned and managed by other teams. This architecture allows you to centralize access to these subgraphs by providing a unified interface and centrally managing authentication, authorization, caching, and more.



When to use this architecture

Technical standpoint

- You want to create a unified/centralized API over multiple subgraphs owned/maintained by your team/s or others.

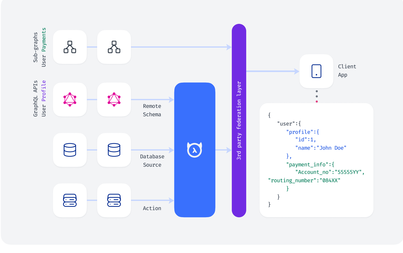
Ownership standpoint

- Your team may or may not own or manage the underlying REST APIs but are allowed to consume and expose the APIs.

Hasura as a subgraph (with Apollo Federation)

Hasura generates a GraphQL spec-compliant schema so Hasura's API can be included as a subgraph in any specification-compliant federation tools.

Hasura also works with some proprietary federation tools. Hasura GraphQL Engine supports the Apollo Federation v1 spec so you can add Hasura as a subgraph in your Apollo federated gateway. You can also use Hasura-generated table types in your other subgraphs by explicitly enabling tables for Apollo Federation.



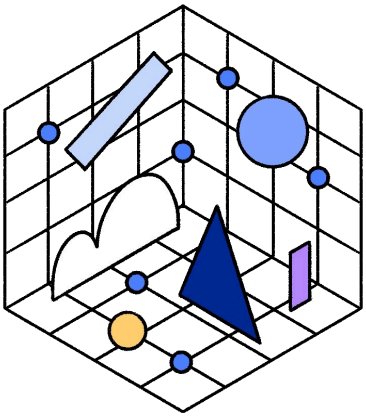
When to use this architecture

Technical standpoint

- You have a Hasura-based GraphQL API and you need to make this API part of a federation layer.

Ownership standpoint

- You own or manage the Hasura-based API and you or another team wants to include this API with others in a single federated API.



Summary

A federation solution empowers developers to build innovative and scalable platforms that address modern API management complexities. In practical terms, this means transforming individual API services into a centralized, accessible point that enhances data access and manipulation across different platforms.

The result is streamlined development by:

- Minimizing the need for middleware
- Using a gateway to expose multiple GraphQL, REST, or gRPC APIs as part of a unified GraphQL endpoint
- Managing subgraphs through a single, authoritative interface

Organizations seeking a seamless federation solution can contact us to explore tailored support and expertise in adopting and managing federated GraphQL APIs.

Glossary

- Subgraph: A GraphQL API that's one of the components in a larger graph. In more practical terms, it's the GraphQL equivalent of a microservice – typically, a GraphQL API built and operated by a single team that owns a domain.
- GraphQL federation: A term introduced by Facebook and popularized by Apollo to describe the process of combining multiple subgraphs into a unified GraphQL API/schema (with relationships between subgraphs).
- Federated GraphQL API: Output of the GraphQL federation process, typically used in the context of using Apollo tooling. Other GraphQL projects also sometimes use this term.
- Supergraph: A generic term used to describe a federated API. The graph in supergraph refers to the combination of API domains rather than just the API itself.

About Hasura

- Hasura lets you effortlessly connect all your databases, services, and code into a unified graph, and expose it via one powerful supergraph API with unparalleled on-demand composability and speed.

- See how Hasura is transforming how businesses, from startups to global enterprises, access and manage data. hasura.io/customers

To speak to our
team reach us on
hasura.io/contact-us

HASURA