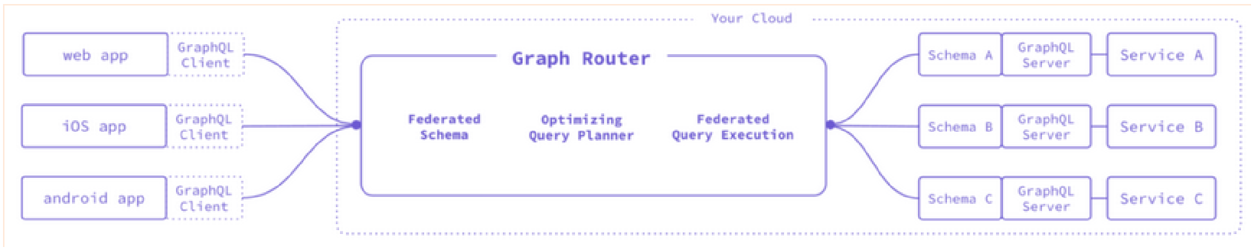
FEDERATION

1. What is Federation

GraphQL Federation is an architecture model that allows multiple GraphQL services, known as subgraphs or federated services, to be combined into a single schema or API.This unified data graph enables clients to query and receive responses from multiple services using a single request.Apollo, a prominent company in the GraphQL ecosystem, pioneered this concept to facilitate the seamless integration of various microservices and data sources.



1.1. Core Principles of GraphQL Federation

Composition: GraphQL Federation enables the composition of multiple, independently deployed GraphQL services into a single schema. This process is facilitated by the Federation Gateway, a server that communicates with each federated service, combines their respective schemas, and resolves incoming queries.

Separation of Concerns: Each federated service can be developed, deployed, and maintained independently, which allows teams to work autonomously and focus on their specific domains.

Schema Extensions: Federation enables the extension of types and fields across services, allowing for seamless data stitching and the resolution of related data from different services.

Query Planning: When a query is received, the gateway generates a query plan, determining the optimal sequence of requests to federated services. This minimizes network overhead and ensures efficient data retrieval.

1.2. Why is Federation Needed

Scalability: Federation enables organizations to scale their GraphQL APIs by distributing the responsibility for specific data domains across multiple services. This simplifies maintenance and allows for independent scaling of individual services.

Flexibility: With the ability to extend types and fields across services, developers can easily add, modify, or remove data sources without affecting the entire schema.

Increased Developer Productivity: The Federation promotes collaboration and accelerates development by separating concerns and allowing teams to work independently on their respective domains.

Easier Data Integration: GraphQL Federation simplifies data integration from multiple sources, making consolidating…

2. Why APIMs need to provide Federation

APIMs cuts out the messy federation process and remove the requirement for you to write all the resolvers yourself. The result? GraphQL expertise at your fingertips without adding an extra service layer to your stack.

Unlike Apollo , APIMs can Scale with confidence. This solution integrates your existing services into a single, GraphQL-enabled endpoint. Simple. Powerful. Highly scalable. Apollo's federation-based approach gets complicated fast. It means you have to add in a whole new service layer or go for a rewrite.

Manage, don't just create - If you're doing GraphQL, you'll need an API management layer eventually. With a APIM , you get everything you need, straight out of the box.

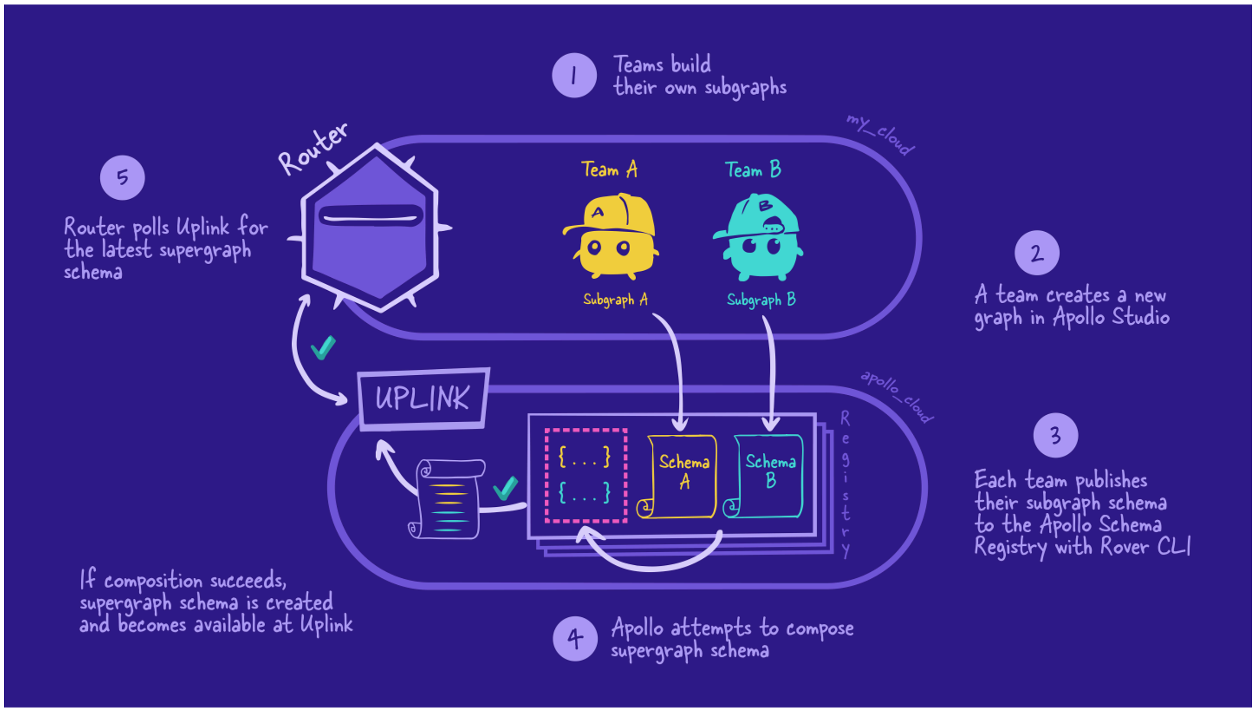
APIM can deliver powerful security that scales with your business, as part of our full lifecycle API management solution.

Makes it easy for you to manage the difference between internal and external use cases for your services. GraphQL proxy lets you limit the schema of the upstream service. It's simple, flexible and effective, with no need for a whole new server or new resolvers.

APIM is easier to use and delivers blistering performance.

Enhanced Developer Experience: Developers benefit from a unified API endpoint, reducing the complexity of working with multiple services and improving overall developer experience.

1. Subgraphs


1
Teams build their own subgraphs
my_cloud
Team A
Team B
Subgraph A
Subgraph B
2
A team creates a new graph in Apollo Studio
3
Each team publishes their subgraph schema to the Apollo Schema Registry with Rover CLI
4
Apollo attempts to compose supergraph schema
5
Router polls Uplink for the latest supergraph schema
Router
UPLINK
apollo_cloud
Registry
Schema A
Schema B
{...}
{...}
If composition succeeds, supergraph schema is created and becomes available at Uplink

Sub graphs ->

GraphQL server with its own schema file, resolvers, and data sources.

Resolvers

responsible for populating the data for a single field in your schema.
A resolver can optionally accept four positional arguments: (parent, args, contextValue, info).

Convert to subgraph servers - import the various directives

```
extend schema
  @link(url: "https://specs.apollo.dev/federation/v2.5",
        import: ["@key"])
```

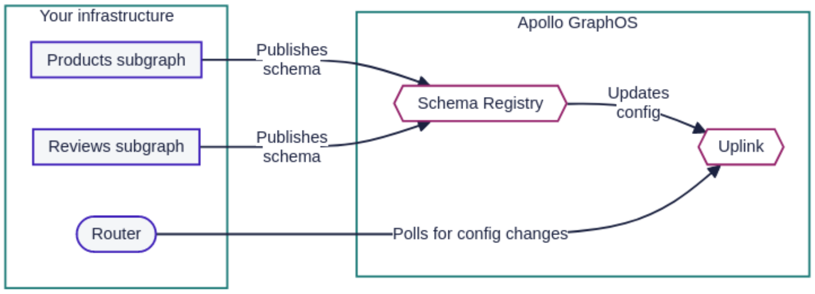
Updating our ApolloServer instance -

Install - @apollo/subgraph
const { buildSubgraphSchema } = require("@apollo/subgraph"); - Makes an object containing typeDefs and resolvers and returns a federation-ready subgraph schema.includes a number of federation directives and types that enable our subgraph to take full advantage of the power of federation

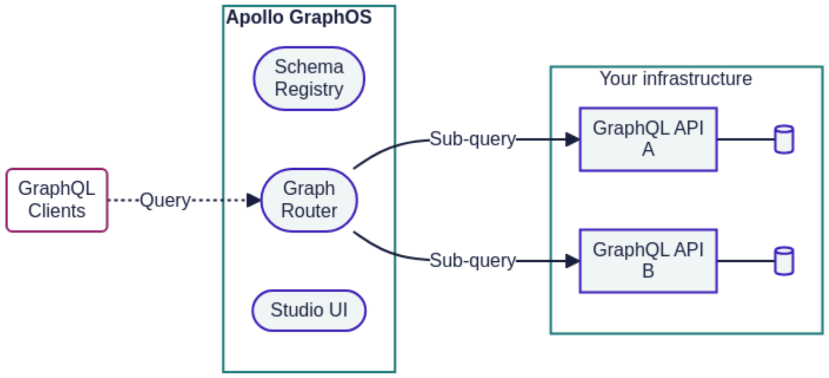
_service
federation-specific fields that buildSubgraphSchema adds to the subgraph. The router uses this field to access the SDL string for your subgraph schema.

ENTITIES
REFERENCING
EXTENDING

Publish sub graphs to GRAPHOS schema registry -> Apollo-hosted version control system, which enables us to track changes to our schemas over time. -> using Rover CLI
Rover is the command-line interface for managing and maintaining graphs with Apollo GraphOS.
MANAGED FEDERATION
Maintain subgraphs and delegate GraphOS to manage CI/CD tasks including the validation, composition, and update of your supergraph
validate, coordinate, deploy, and monitor changes to your graph
Router stability -> no downtime in updates
Composition stability -> provide the most recent valid configuration to your router.
Schema flexibility -> ensure the safety of certain schema changes

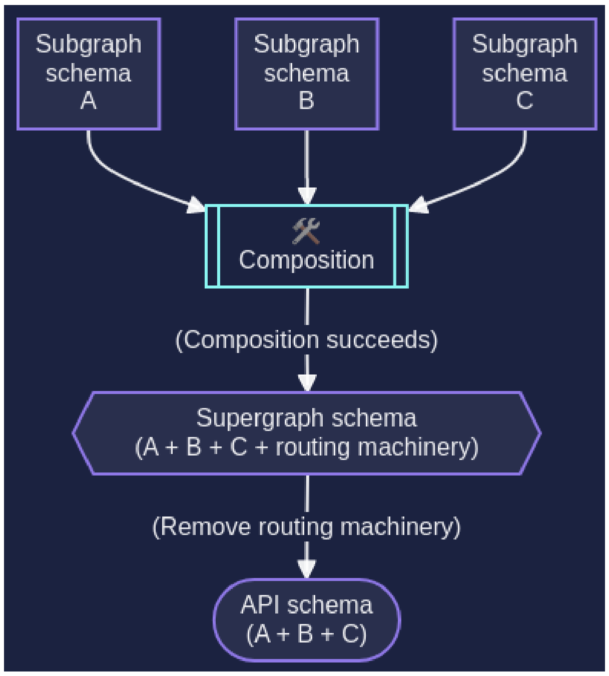


GRAPH OS
platform for building, managing, and scaling a supergraph



2. Supergraph Composition

combine all of the schemas from the registered subgraphs into a single supergraph schema. By SCHEMA Registry



- Subgraph schemas - distinct schema indicating types and fields supergraph can resolve.
  - These are the only schemas that your teams define manually.
- Supergraph schema - combines all types and fields from subgraph schemas, *plus* some federation-specific information that tells router which subgraphs can resolve which fields.
  - Result of performing composition on subgraph schemas.
- API schema - Similar to the supergraph schema - *omits* federation-specific types, fields, and directives not part of your public API.
  - Schema router expose to clients

Subgraphs compose to form a supergraph using federated directives

1. **@key Directive**: Fundamental. Annotate a type in subgraph representing unique identifier for type. The `@key` directive specifies one or more fields that together form a unique key for instances of the type.
2. **@requires and @provides Directives**: Specify dependencies between fields within a subgraph. The `@requires` - particular field depends on the values of other fields, while the `@provides` - field provides values that can be used by other fields.

supergraph's resolvers utilize unique keys to fetch the referenced data from the appropriate subgraph.

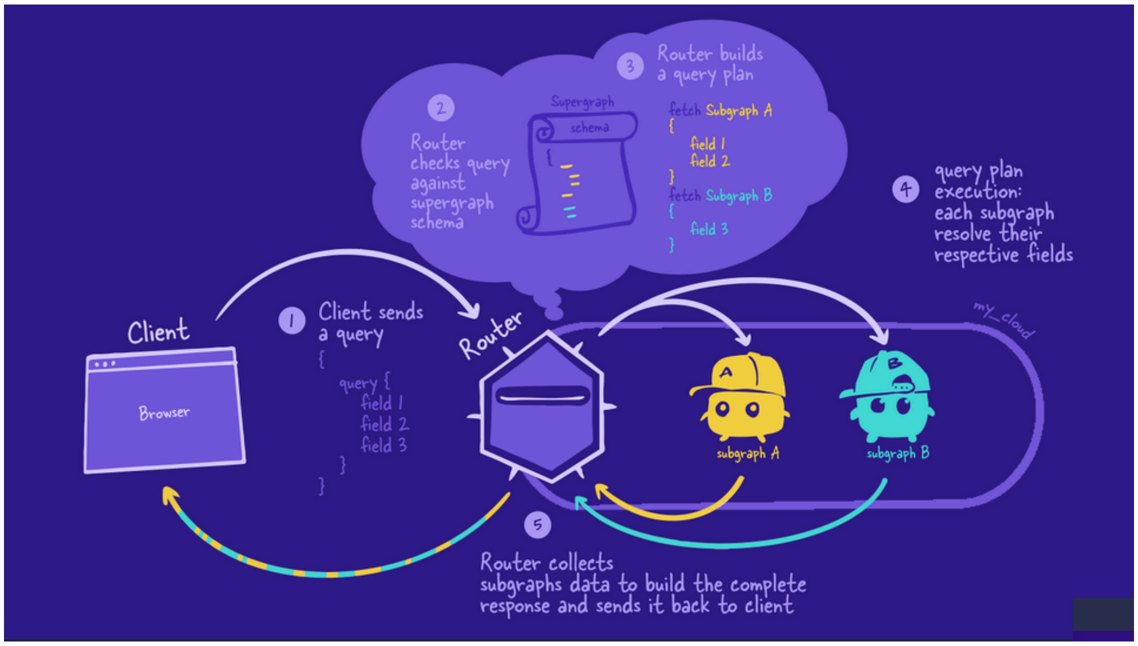
Convert to subgraph servers - import the various directives
extend schema
  @link(url: "https://specs.apollo.dev/federation/v2.5",
          import: ["@key"])

3. Query Builder

A query plan is a blueprint for dividing a single incoming operation into one or more operations that are each resolvable by a single subgraph. Some of these operations depend on the results of other operations, so the query plan also defines any required ordering for their execution.

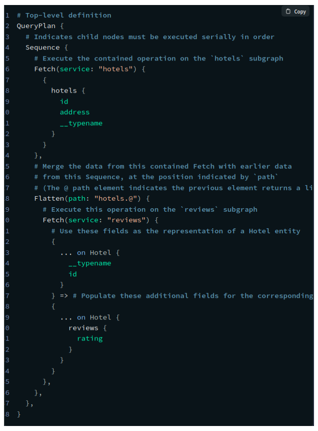
With managed federation, schema updates to the router are managed by Apollo Studio and happen with zero downtime.

Routing



Router builds a query plan
Supergraph schema
fetch Subgraph A { field 1 field 2 }
fetch Subgraph B { field 3 }
Router checks query against supergraph schema
query plan execution: each subgraph resolve their respective fields
Client
Browser
Client sends a query
{ query { field 1 field 2 field 3 } }
Router
my_cloud
subgraph A
subgraph B
Router collects subgraphs data to build the complete response and sends it back to client

1. Client Sends Query
2. Router checks supergraph Schema -> resolve which subgraph responsible for which field - Identify dependencies
3. Router builds Query plan



```
# Top-level definition
QueryPlan {
  # Indicates child nodes must be executed serially in order
  Sequence {
    # Execute the contained operation on the `hotels` subgraph
    Fetch(service: "hotels") {
      {
        hotels {
          id
          address
          __typename
        }
      }
    },
    # Merge the data from this contained Fetch with earlier data
    # from this Sequence, at the position indicated by `path`
    # (The @ path element indicates the previous element returns a li
    Flatten(path: "hotels.@") {
      # Execute this operation on the `reviews` subgraph
      Fetch(service: "reviews") {
        # Use these fields as the representation of a Hotel entity
        {
          ... on Hotel {
            __typename
            id
          }
        } => # Populate these additional fields for the corresponding
        {
          ... on Hotel {
            reviews {
              rating
            }
          }
        }
      },
    },
  },
}
```

Blueprint for dividing a single incoming operation into one or more operations that are each resolvable by a single subgraph. Some of these operations depend on the results of other operations, so the query plan also defines any required ordering for their execution.

hierarchy of nodes that looks like a JSON or GraphQL document when serialized

Each node defined inside the QueryPlan node is one of the following:

Fetch - Tells the gateway to execute a particular operation on a particular subgraph.
Parallel - Tells the gateway that the node's immediate children can be executed in parallel.independent operations
Sequence - Tells the gateway that the node's immediate children must be executed serially in the order listed. *one* subgraph's response depends on data that first must be returned by *another* subgraph
Flatten - Tells the gateway to merge the data returned by this node's child Fetch node with data previously returned in the current Sequence.

4. Query Plan Execution : router fetch from each subgraph and each subgraph resolve their respective fields normally use their resolvers and data sources to retrieve and populate the requested data.
   uses the federated directives and the schema definitions to merge the data from different subgraphs, resolving references between them as necessary.

5. Router collects subgraphs data to build the complete response and sends it back to the client - The subgraphs send back the requested data to the router, and then the router combines all those responses into a single JSON object.

4. Query Execute
5. Response

https://hasura.io/blog/client-side-graphql-schema-resolving-and-schema-stitching-f4d8bccc42d2/

https://hygraph.com/blog/schema-stitching-vs-graphql-federation-vs-content-federation

GRAPHQL Federation

Maven
- build automation and project management tool
- structure projects, manage dependencies, and perform various tasks related to building, testing, and packaging software.
- Pom.xml - specifies the dependencies for maven
- Maven - resolve retrieve dependencies
- Resolve dependency trees (transitive)

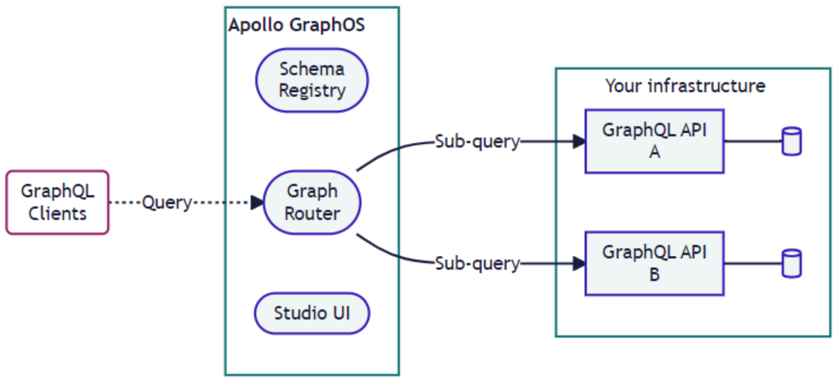
Build Project

- transforming source code and resources into executable or deployable artifacts.
- compiling source code, resolving and downloading dependencies, running tests, and creating distributable packages.

GraphQl

- Alternative to REST
- Operations - query,mutation,subscription
-

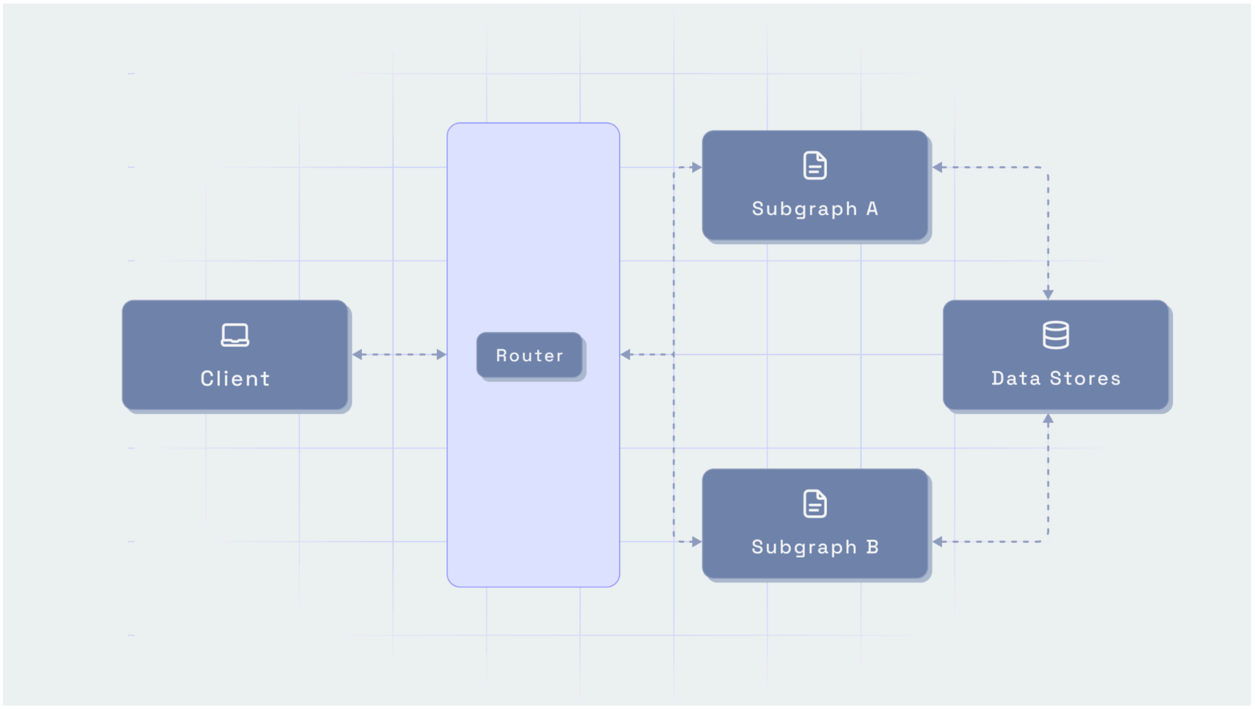
Unifying multiple APIs



Service Registration

Resolvers

Resolvers are responsible for fetching data for the fields in the schema, and they are implemented for each type and field in the schema



Links
https://stepzen.com/blog/schema-stitching-and-federation-not-the-best-solutions-for-graphql-microservices

https://medium.com/booking-com-development/improving-graphql-federation-resiliency-ca3e95075de4

https://medium.com/paypal-tech/graphql-resolvers-best-practices-cd36fdbcef55

https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2

https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2

Build - background federation

`GraphQL federation

introspectAndCompose

- dynamically fetching and composing the schemas from multiple federated services into a unified supergraph
- Introspection: The introspectAndCompose method starts by performing schema introspection on each service defined in the serviceList. It sends an introspection query to each service's GraphQL endpoint to retrieve its schema definition.
- Schema Composition: Once the schemas are introspected, the introspectAndCompose method proceeds to compose these schemas into a single, unified schema known as the supergraph. This involves merging types, resolving conflicts, and establishing relationships between types across different services.
- Field Resolution: The method also resolves fields in the supergraph by traversing the relationships and dependencies defined by the @key, @extends, @provides, and @requires directives in the schemas. This allows it to determine how data should be fetched from multiple services and combined into a cohesive response.

Apollo Gateway
central component responsible for orchestrating the federation of multiple GraphQL services into a single, unified GraphQL API.
Initialization: When an instance of ApolloGateway is created, it is typically provided with a configuration object that includes a list of services. Each service is defined by a name and a URL to its GraphQL endpoint.
Schema Loading: Upon initialization, the ApolloGateway class begins the process of loading the schemas from the specified services. It uses the introspectAndCompose method (or similar mechanisms) to fetch the schemas, perform schema introspection, and compose them into a unified supergraph schema.

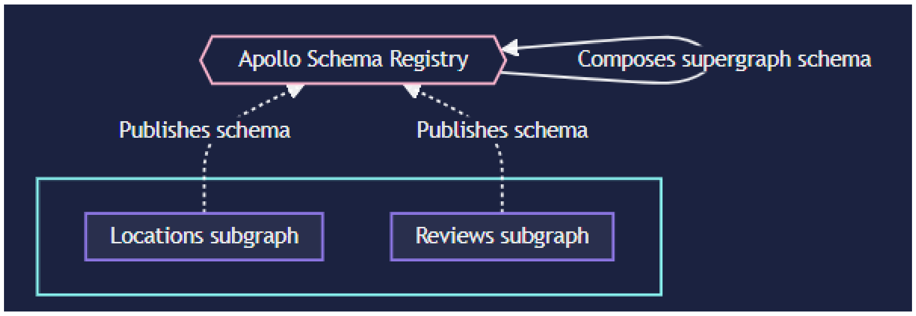
Breaking composition
- conflict in a way that causes composition to fail

Rules of composition
- Multiple subgraphs can't define the same field on an object type, unless that field is shareable.
- A shared field must have both a compatible return type and compatible argument types across each defining subgraph.
    - For examples of compatible and incompatible differences between subgraphs, see Differing shared fields.
- If multiple subgraphs define the same type, each field of that type must be resolvable by every valid GraphQL operation that includes it.

1. Multiple Services
2. @key -> directive to specify which is globally unique
3. __resolveReference -> how to fetch data for entity

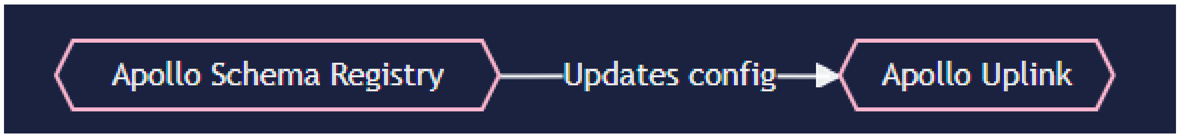
4. Gateway -> introspects schemas

Supergraph compose by managed federation

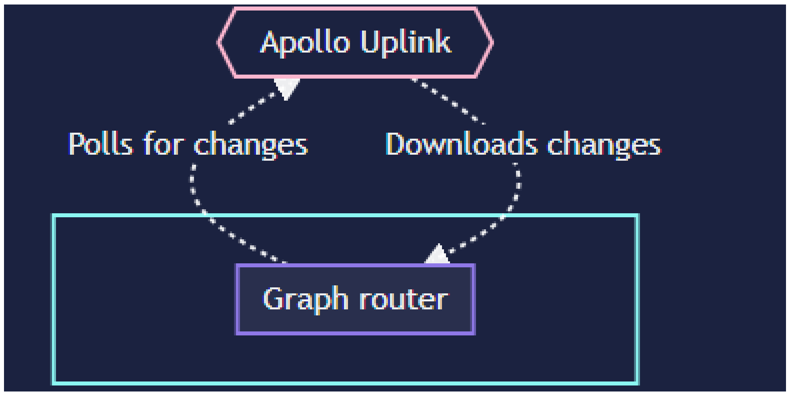


\

Apollo Uplink - if composition succeed



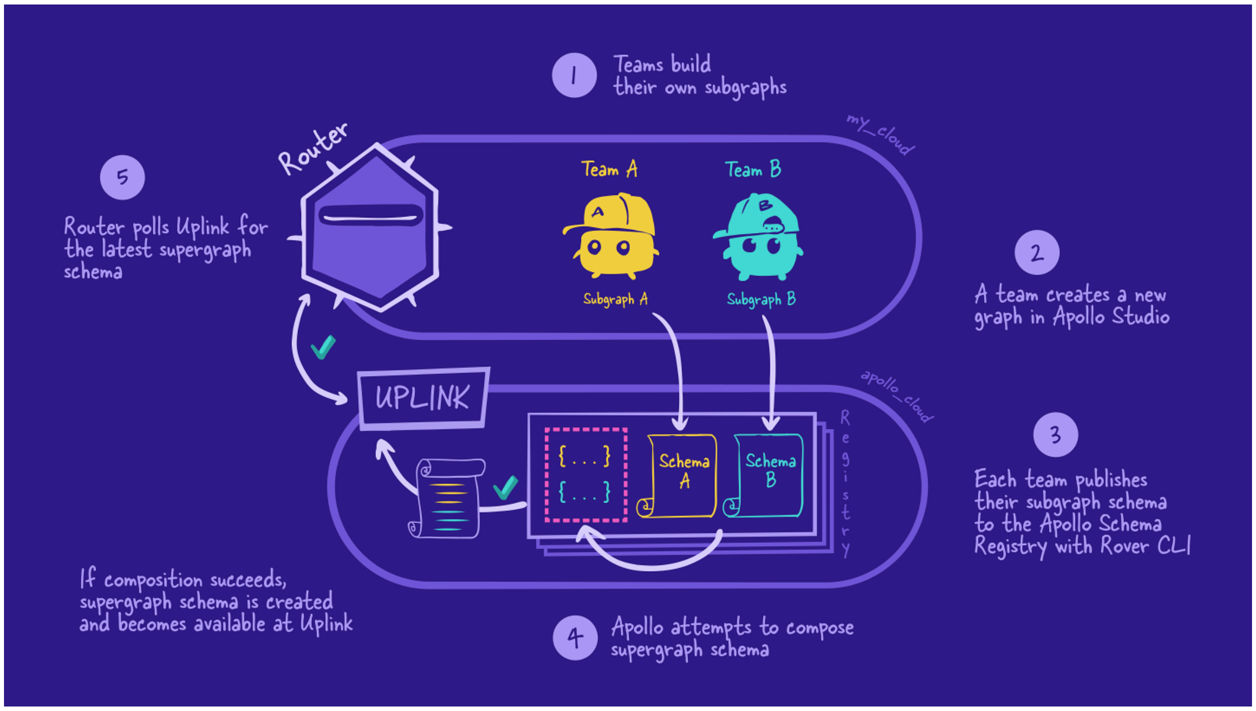
Router polls for changes



managed federation for reducing downtime

reference resolver tells the gateway how to fetch an entity by its `@key` fields.

router
query plan building
query plan execution
subservices fetching mechanism
response building
response streaming


1
Teams build their own subgraphs
my_cloud
Team A
Team B
Subgraph A
Subgraph B
2
A team creates a new graph in Apollo Studio
3
Each team publishes their subgraph schema to the Apollo Schema Registry with Rover CLI
apollo_cloud
Registry
Schema A
Schema B
{ . . . }
{ . . . }
4
Apollo attempts to compose supergraph schema
If composition succeeds, supergraph schema is created and becomes available at Uplink
UPLINK
Router
5
Router polls Uplink for the latest supergraph schema

1. Sub graphs ->
   GraphQL server with its own schema file, resolvers, and data sources.

Resolvers
responsible for populating the data for a single field in your schema.
A resolver can optionally accept four positional arguments: (parent, args, contextValue, info).

Convert to subgraph servers - import the various directives
extend schema
  @link(url: "https://specs.apollo.dev/federation/v2.5",
        import: ["@key"])

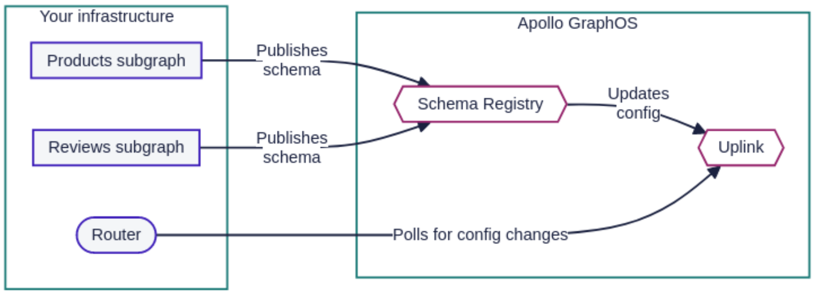
Updating our ApolloServer instance -
Install - @apollo/subgraph
const { buildSubgraphSchema } = require("@apollo/subgraph"); -akes an object containing typeDefs and resolvers and returns a federation-ready subgraph schema.ncludes a number of federation directives and types that enable our subgraph to take full advantage of the power of federation

_service
ederation-specific fields that buildSubgraphSchema adds to the subgraph. The router uses this field to access the SDL string for your subgraph schema.

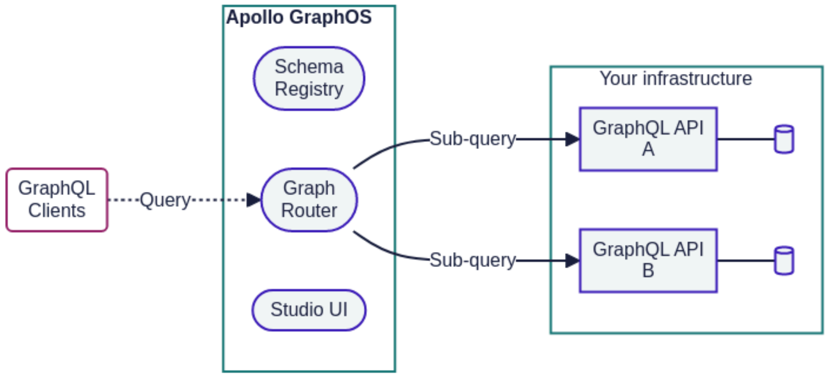
ENTITIES

REFERENCING
EXTENDING

2. Publish sub graphs to GRAPHOS schema registry -> Apollo-hosted version control system, which enables us to track changes to our schemas over time. -> using Rover CLI
   Rover is the command-line interface for managing and maintaining graphs with Apollo GraphOS.
   <u>MANAGED FEDERATION</u>
   Maintain subgraphs and delegate GraphOS to manage CI/CD tasks including the validation, composition, and update of your supergraph
   validate, coordinate, deploy, and monitor changes to your graph
      Router stability -> no downtime in updates
      Composition stability -> provide the most recent valid configuration to your router.
      Schema flexibility -> ensure the safety of certain schema changes



   <u>GRAPH OS</u>
   platform for building, managing, and scaling a supergraph

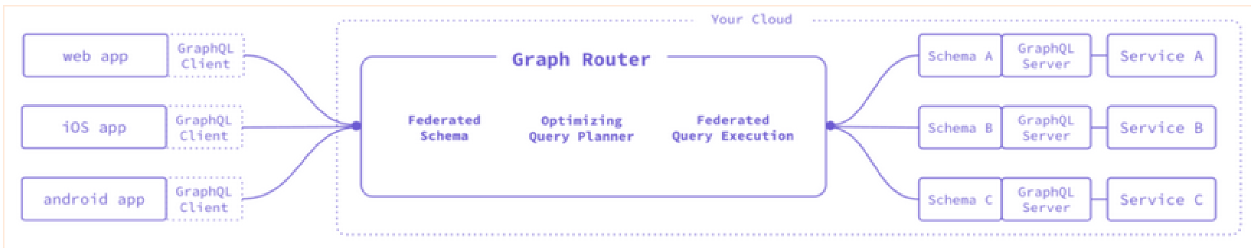


3. Composition - > combine all of the schemas from the registered subgraphs into a single supergraph schema. By SCHEMA Registry
4. Apollo Uplink -> tSores the latest supergraph schema for each graph -> sent by schema registry

5. Router -> resolve requests
    a. If changes super graph -> update no downtime
    b. If change update and use

https://www.apollographql.com/blog/apollo-router-our-graphql-federation-runtime-in-rust

Rust 8X faster than JS



A query plan is a blueprint for dividing a single incoming operation into *one or more* operations that are each resolvable by a single subgraph. Some of these operations depend on the results of *other* operations, so the query plan also defines any required ordering for their execution.

With managed federation, schema updates to the router are managed by Apollo Studio and happen with zero downtime.

Server , gateway and subgraphs

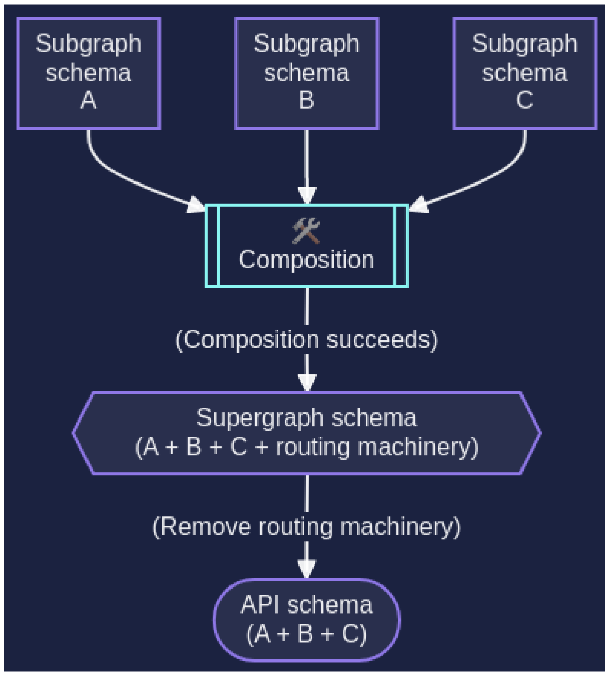
Hasura

- Automating almost 50-80% if the federation

When considering Hasura vs Apollo, it's important to note that Hasura provides a more out-of-the-box solution for connecting to multiple data sources, whereas Apollo requires more manual setup for schema stitching and federation.

- For the field extended with the remote join, Hasura will pass the value of the selected local field as an argument to the remote service's resolver.
- The remote service's resolver will then use this argument value to fetch and return the relevant data.
- Hasura metadata - point of collaboration

Composition



- Subgraph schemas - distinct schema indicating types and fields supergraph can resolve.
    - These are the only schemas that your teams define manually.
- Supergraph schema - combines all types and fields from subgraph schemas, *plus* some federation-specific information that tells router which subgraphs can resolve which fields.
    - Result of performing composition on subgraph schemas.
- API schema - Similar to the supergraph schema - *omits* federation-specific types, fields, and directives not part of your public API.
    - Schema router expose to clients

Subgraphs compose to form a supergraph using [federated directives]

1. **@key Directive**: Fundamental. Annotate a type in subgraph representing unique identifier for type. The `@key` directive specifies one or more fields that together form a unique key for instances of the type.
2. **@requires and @provides Directives**: Specify dependencies between fields within a subgraph. The `@requires` - particular field depends on the values of other fields, while the `@provides` - field provides values that can be used by other fields.

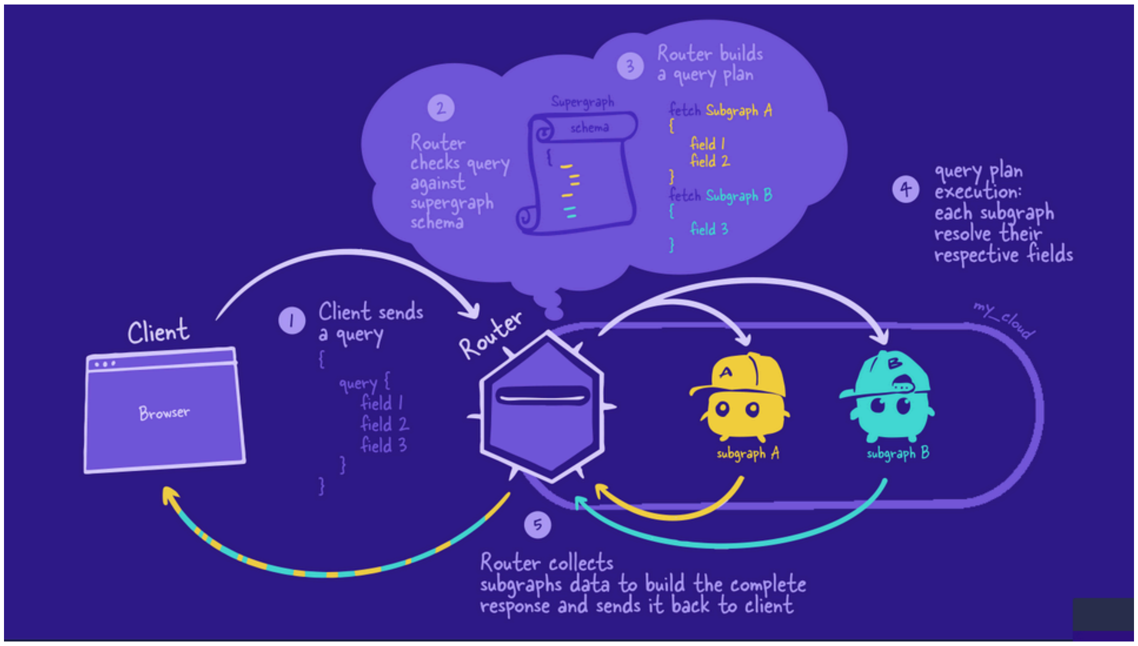
supergraph's resolvers utilize unique keys to fetch the referenced data from the appropriate subgraph.

Convert to subgraph servers - import the various directives

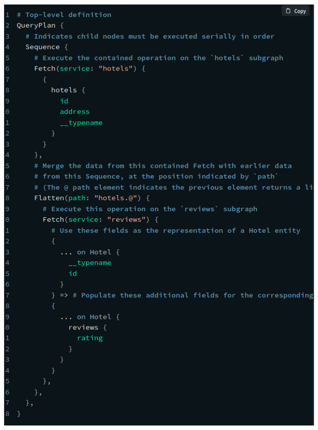
```
extend schema
  @link(url: "https://specs.apollo.dev/federation/v2.5",
          import: ["@key"])
```

Managed Federation

Routing


Client
Browser
1 Client sends a query
{
query {
field 1
field 2
field 3
}
}
Router
2 Router checks query against supergraph schema
Supergraph schema
3 Router builds a query plan
fetch Subgraph A
{
field 1
field 2
}
fetch Subgraph B
{
field 3
}
4 query plan execution: each subgraph resolve their respective fields
my_cloud
subgraph A
subgraph B
5 Router collects subgraphs data to build the complete response and sends it back to client

1. Client Sends Query
2. Router checks supergraph Schema -> resolve which subgraph responsible for which field - Identify dependencies
3. Router builds Query plan



```
# Top-level definition
QueryPlan {
  # Indicates child nodes must be executed serially in order
  Sequence {
    # Execute the contained operation on the `hotels` subgraph
    Fetch(service: "hotels") {
      {
        hotels {
          id
          address
          __typename
        }
      }
    },
    # Merge the data from this contained Fetch with earlier data
    # from this Sequence, at the position indicated by `path`
    # (The @ path element indicates the previous element returns a li
    Flatten(path: "hotels.@") {
      # Execute this operation on the `reviews` subgraph
      Fetch(service: "reviews") {
        # Use these fields as the representation of a Hotel entity
        {
          ... on Hotel {
            __typename
            id
          }
        } => # Populate these additional fields for the corresponding
        {
          ... on Hotel {
            reviews {
              rating
            }
          }
        }
      },
    },
  },
}
```

Blueprint for dividing a single incoming operation into one or more operations that are each resolvable by a single subgraph. Some of these operations depend on the results of other operations, so the query plan also defines any required ordering for their execution.

hierarchy of nodes that looks like a JSON or GraphQL document when serialized

Each node defined inside the QueryPlan node is one of the following:

Fetch - Tells the gateway to execute a particular operation on a particular subgraph.
Parallel - Tells the gateway that the node's immediate children can be executed in parallel.independent operations
Sequence - Tells the gateway that the node's immediate children must be executed serially in the order listed. *one* subgraph's response depends on data that first must be returned by *another* subgraph

Flatten - Tells the gateway to merge the data returned by this node's child Fetch node with data previously returned in the current Sequence.

4. Query Plan Execution : router fetch from each subgraph and each subgraph resolve their respective fields normally use their resolvers and data sources to retrieve and populate the requested data.
   uses the federated directives and the schema definitions to merge the data from different subgraphs, resolving references between them as necessary.

5. Router collects subgraphs data to build the complete response and sends it back to the client - The subgraphs send back the requested data to the router, and then the router combines all those responses into a single JSON object.
6.

QUERY PLANNER

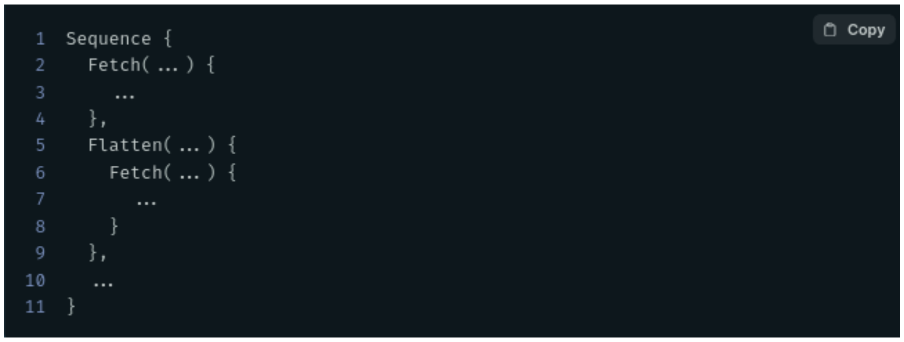
```
# Top-level definition
QueryPlan {
 # Indicates child nodes must be executed serially in order
 Sequence {
   # Execute the contained operation on the `hotels` subgraph
   Fetch(service: "hotels") {
     {
       hotels {
         id
         address
         __typename
       }
     }
   },
   # Merge the data from this contained Fetch with earlier data
   # from this Sequence, at the position indicated by `path`
   # (The @ path element indicates the previous element returns a list)
   Flatten(path: "hotels.@") {
     # Execute this operation on the `reviews` subgraph
     Fetch(service: "reviews") {
       # Use these fields as the representation of a Hotel entity
       {
         ... on Hotel {
           __typename
           id
         }
```

```
        } => # Populate these additional fields for the corresponding Hotel
        {
          ... on Hotel {
            reviews {
              rating
            }
          }
        }
      },
    },
  },
}
```

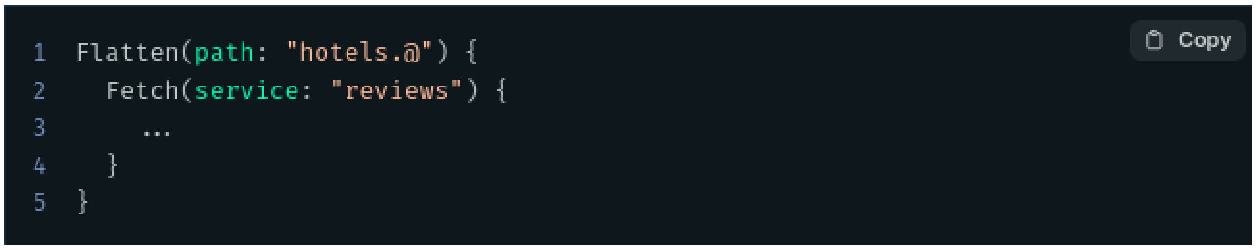
hierarchy of nodes that looks like a JSON or GraphQL document when serialized
At the top level of every query plan is the QueryPlan node:

Each node defined inside the QueryPlan node is one of the following:
Fetch
- Tells the gateway to execute a particular operation on a particular subgraph.

Parallel
- Tells the gateway that the node's immediate children can be executed in parallel.
- appears in query plans whenever the router can execute completely independent operations on different subgraphs.

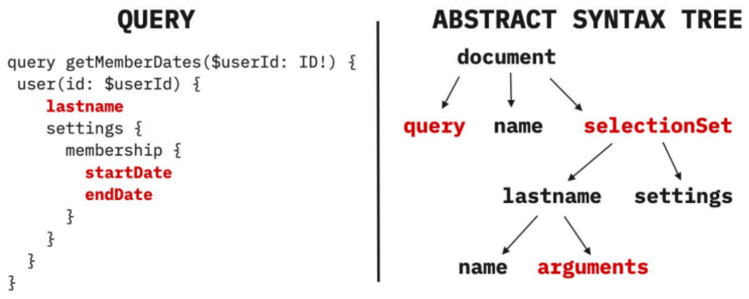
Sequence
- Tells the gateway that the node's immediate children must be executed serially in the order listed.
- ppears in query plans whenever one subgraph's response depends on data that first must be returned by another subgraph.
- 
```
Sequence {
  Fetch( ... ) {
    ...
  },
  Flatten( ... ) {
    Fetch( ... ) {
      ...
    }
  },
  ...
}
```

Flatten
- Tells the gateway to merge the data returned by this node's child Fetch node with data previously returned in the current Sequence.
- @ element in a path indicates that the immediately preceding path element returns a list.

- path argument tells the router at what position to merge the newly returned data with the existing data
- 

```
Flatten(path: "hotels.@") {
  Fetch(service: "reviews") {
    ...
  }
}
```

1. **Parsing the Query:**
   - Breaking down the query string into a structured abstract syntax tree (AST), which represents the structure of the query.



```
query getMemberDates($userId: ID!) {
 user(id: $userId) {
    lastname
    settings {
      membership {
        startDate
        endDate
      }
    }
  }
}
```

2. **Identifying Special Fields:**
   - During the parsing process, the federation gateway identifies special fields that require federated handling. In this example, the special field is `_entities`.
3. **Processing `_entities` Field:**
   - When the `_entities` field is encountered, the gateway extracts the representations of entities specified in the query. The representations include the entity type (`__typename`) and the ID of each entity.
4. **Routing to Respective Services:**
   - For each entity representation, the gateway determines the service responsible for handling that entity type. This is based on the federation configuration and the `@key` directives in each service's schema.
5. **Creating Subqueries:**

    - The gateway creates subqueries for each service, targeting the `_entities` field in the respective service's schema. Each subquery specifies the entity type and ID for that service.

6. **Sending Subqueries to Services:**
    - The gateway dispatches the generated subqueries to the corresponding services. Each service processes its subquery and returns the requested data.
7. **Aggregating Results:**
    - Once the responses from individual services are received, the gateway aggregates the results. It combines the data for each entity type into a unified response.
8. **Returning Final Response:**
    - The query plan is now complete. The federated gateway returns the final response to the client, which includes the data for all requested entities.

https://planetscale.com/blog/what-is-a-query-planner

https://docs.shiftleft.io/core-concepts/c-syntaxtree

https://www.linkedin.com/pulse/graphql-query-execution-how-powers-modernapis-tabish-manzoor-1f/

https://adamhannigan81.medium.com/understanding-the-graphql-ast-f7f7b8e62aa4

Query Plannner

- Semantic analysis

  Bind to tables and columns
  Which table a column comes from, what types the columns and expressions in the query have

- Optimization

  determine the most efficient way to execute the query
  selecting the most efficient algorithms for operations such as joins and sorting, and choosing the most appropriate indexes to use
  All the tables need to be visited, and the connections between tables have different costs. The planner can start at any (*) node. What is the path that touches all tables with the least cost? This is why the join order is so important.

- Code generation

  exact steps that the database engine should take to execute the query.

1. **Query Submission**

When you send a GraphQL query, you make a single HTTP POST request to the GraphQL server's endpoint, typically /graphql. The request body contains the query you want to execute. Here's an example of a GraphQL query:

```
query {
 user(id: 1) {
 name
 email
 }
 }
```

2. **Schema Validation**

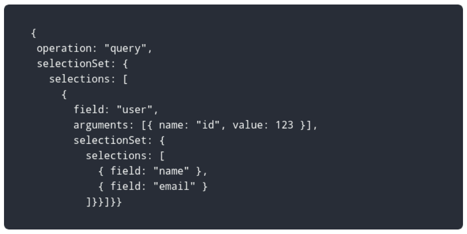
The GraphQL server has a defined schema that specifies the available types, queries, mutations, and their structure. It serves as the contract between the client and server. Before processing the query, the server validates it against the schema to ensure it conforms to the expected structure. GraphQL combines the query document that the user requested with the schema definition that we defined for our resolver in the form of an AST. This AST is used to determine which fields were requested, what arguments were included and much more.

3. **Query Parsing**

The server parses the incoming query to understand its structure. Parsing contains the following steps.

1- Lexical Analysis (Tokenization): The first step in parsing is lexical analysis, also known as tokenization. The query string is broken down into a stream of tokens, which are the smallest units of the query.

2- Syntax Parsing: The token stream is then parsed to construct an abstract syntax tree (AST). This tree represents the hierarchical structure of the query and is used to validate the query's syntax.



```
{
 operation: "query",
 selectionSet: {
   selections: [
     {
       field: "user",
       arguments: [{ name: "id", value: 123 }],
       selectionSet: {
         selections: [
           { field: "name" },
           { field: "email" }
         ]}}]}}
```

. . .

4. **Execution Planning**

GraphQL generates an execution plan by traversing the AST and resolving each field based on the parsed query.

A query plan serves as a strategic framework for decomposing an initial operation(query receive from user) into several sub-operations, each of which can be independently resolved within its own subgraph.

5. **Field Resolution**

GraphQL resolves each field in the query by invoking a corresponding resolver function. When a field is executed, the corresponding resolver is called to produce the next value. If a field produce a scalar value (number or string) then the execution completes. However if a field produce object then query continue until scalar values are reached.

6. **Data Fetching**

Resolvers may need to fetch data from one or more data sources. GraphQL allows parallel execution of resolvers, which means that data retrieval can be optimized for efficiency. This is in contrast to REST, where you often have to make multiple sequential requests to different endpoints to gather related data.

7. **Data Stitching and response generation**

As data is retrieved, GraphQL stitches the results together into a single JSON response object. The response structure matches the structure of the query . The GraphQL server sends the response back to the client in JSON format

Cost function is likely used to evaluate and compare different query plans, helping the system choose the most efficient plan for executing a GraphQL query.

SelectionIsFullyLocalFromAllVertices checks whether a given GraphQL selection set is fully local from a set of vertices in a federated GraphQL setup.

compareOptionsComplexityOutOfContext Comparing the complexity of two sets of simultaneous paths

Subgraphs

- Validate / update to federation ready graphs

Supergraphs

generateAllPlansAndFindBest

Start

```
|---> Initialize QueryPlanningTraversal
|---> Receive query input
|---> Set initial state (open branches, cost function)

Loop: While open branches exist
|---> Select a branch from open branches
|---> Explore branch (depth-first traversal)
|---> Expand potential fetch operations
        Start (Expand Potential Fetch Operations)

        |---> Identify potential fetch targets:
        |---> Analyze query fields and relationships
        |---> Determine which subgraphs hold required data

        |---> For each potential target:
        |---> Consider fetch options:
        |---> Fetch entire object: Retrieve all fields from subgraph.
        |---> Fetch specific fields: Retrieve only necessary fields.
        |---> Eager fetch related data: Pre-fetch data from related objects.
        |---> Defer fetching (if possible): Delay fetching until later.

        |---> Evaluate cost of each option:
        |---> Apply cost function (network fetches, field resolution, etc.)

        |---> Choose lowest-cost options:
        |---> Create new branches for chosen fetch operations
        |---> Add branches to open branches list

        End
|---> Evaluate cost of each operation
|---> Choose lowest-cost operation
|---> Update state (fetch results, open/closed branches)
|---> Handle type conditions (fetch __typename if needed)
|---> Debug/Log planning progress (optional)

End Loop

|---> Select best plan from closed branches
|---> Return optimized query plan
```

https://medium.com/building-the-open-data-stack/build-a-crypto-app-using-graphql-federation-56aca9a3dde1

QOS to federation

https://konghq.com/solutions/api-management-graphql

Supergraph Layer:

- **Subgraph Traffic Shaping**: At the supergraph layer, traffic shaping policies can be applied to individual subgraphs. This includes enforcing rate limits and timeouts specific to each subgraph based on its workload and capacity.

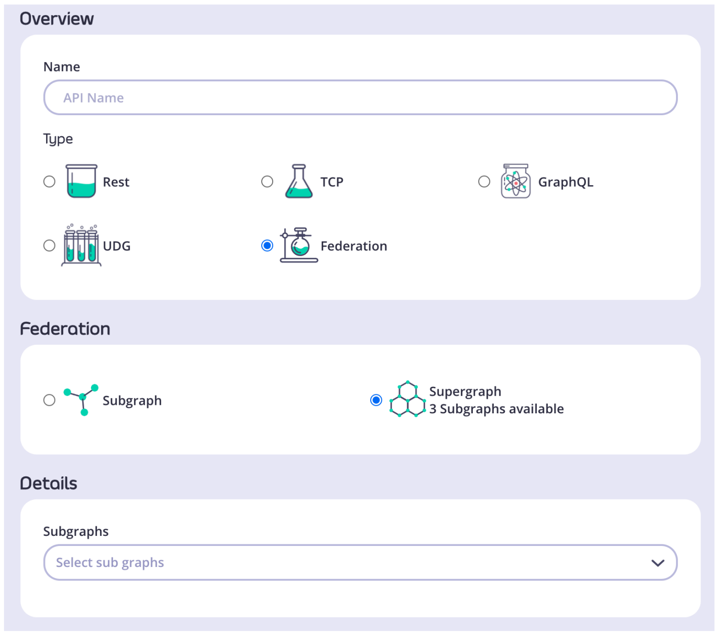
https://github.com/apollographql/federation-jvm/blob/main/graphql-java-support/README.md

WHY APIM to handle Federation -> Manage, don't just create

- Developer experience
- Performance - better than apollo
- Better Experience in federation cutting out the Upstream schema change
- Manage and control access to your GraphQL services for both internal and external use cases - limit the schema of the upstream service
- Providing a visual interface for stitching together data from multiple sources into a single GraphQL API.
- GUI to turn it into a single API. No new code, services or infrastructure required.
- Secure, flexible, scalable : Apollo ->Harder to scale

Competitors

Tyk
https://tyk.io/docs/getting-started/key-concepts/graphql-federation/



Overview

Name

API Name

Type

○ Rest ○ TCP ○ GraphQL

○ UDG [✓ Federation]

Federation

○ Subgraph [✓ Supergraph 3 Subgraphs available]

Details

Subgraphs

Select sub graphs

Federation -> Subgraph > SUpergraph
Git Repo -> https://github.com/TykTechnologies/tyk.git
Universal Data Graph

https://tyk.io/alternatives-apollo/

Subscriptions -> Apollo Support

Kong -> Check federation
MuleSoft , Azure

https://konghq.com/solutions/api-management-graphql

- subgraphs 3 ()
- Schema -> combine -> how to expose in gateway with supergraph
- How supergraph
- Write graphql service -> expose supergraph (unifed graphql schema)
- Api invoke -> query
- Query payload - unified schema

- Validation ->

https://konghq.com/solutions/api-management-graphql

https://dev.to/graphqleditor/graphql-java-cheat-sheet-4316
https://github.com/apollographql/federation-jvm
https://www.ibm.com/docs/en/api-connect/ace/1.x?topic=directives-building-supergraph
https://www.apollographql.com/docs/federation/building-supergraphs/supported-subgraphs/

In the example above the GraphQL `Show` type name maps to the Java `Show` type. There are also cases where the GraphQL and Java type names don't match, specially when working with existing code. If any of your class names do not match your schema type names, you need to provide this class with a way to map between them. To do this, return a map from the `typeMapping()` method in your own implementation of the `DefaultDgsFederationResolver`. In the following example we map the GraphQL `Show` type to a `ShowId` Java type.

this seamless developer experience enhances productivity, fosters collaboration, and accelerates the development and adoption of APIs within organizations and their ecosystems
1. Unified Management and Governance:
Centralized Control: By offering GraphQL federation capabilities within an APIM, organizations can manage and govern both RESTful and GraphQL APIs from a single platform. This centralized approach simplifies management, monitoring, security, compliance, and other aspects of API governance, ensuring consistency and control across the entire API landscape.

2. Integrated Security and Compliance:
Robust Security Features: APIMs like Apigee provide built-in security features, such as OAuth, JWT, API key validation, rate limiting, and data masking. By integrating GraphQL federation capabilities within an APIM, organizations can enforce consistent security policies and compliance requirements across both RESTful and GraphQL APIs, without the need for complex integration or additional security layers.

3. Scalability and Performance:
Optimized Infrastructure: APIMs are designed to handle high volumes of API traffic, ensuring scalability, performance, and reliability. By offering GraphQL federation capabilities within an APIM, organizations can benefit from the optimized infrastructure components of the APIM, ensuring that federated GraphQL services can scale and perform efficiently.

4. Developer Experience and Productivity:
Integrated Developer Portals: APIMs offering GraphQL federation capabilities typically include integrated developer portals, documentation, and API catalogs. This seamless developer experience enhances productivity, fosters collaboration, and accelerates the development and adoption of APIs within organizations and their ecosystems.

5. Ecosystem Integration and Collaboration:
Unified Ecosystem: Offering GraphQL federation capabilities within an APIM enables seamless integration and collaboration within the organization's ecosystem, including partners, suppliers, and third-party developers. This collaborative approach fosters innovation, accelerates time-to-market, and creates new revenue opportunities through partnerships and ecosystem integration

Universal Data Graph (UDG),

Tyk Vs APOLLO

The Tyk API Gateway and Universal Data Graph are fast. Incredibly fast.

Tyk cuts out the messy federation process and removes the requirement to write all the resolvers yourself.

Tyk's advanced Universal Data Graph provides ultimate flexibility. This next-gen solution integrates your existing services into a single, GraphQL-enabled endpoint. Simple. Powerful. Highly scalable.

Apollo's federation-based approach gets complicated fast. It means you have to add in a whole new service layer or go for a rewrite.

Manage, don't just create
If you're doing GraphQL, you'll need an API management layer eventually. With Tyk, you get everything you need, straight out of the box. -> developer Experience

Tyk makes it easy for you to manage the difference between internal and external use cases for your services. Our GraphQL proxy lets you limit the schema of the upstream service. It's simple, flexible and effective, with no need for a whole new server or new resolvers.

With Apollo, you have to write everything in NodeJS. Developers use their OSS libraries to write the GraphQL server, which itself is a bridge to their other services (REST, database, etc.). This means every service needs to be re-created in a GraphQL representation inside the GraphQL server to make it work.

That sounds like a lot of work to us. With Tyk's Universal Data Graph, you can simply stitch together all of your disparate data from multiple sources. Use the Tyk GUI to turn it into a single API. No new code, services or infrastructure required.

https://tyk.io/blog/an-introduction-to-graphql-federation/
https://hygraph.com/blog/schema-stitching-vs-graphql-federation-vs-content-federation
https://www.apollographql.com/docs/federation/migrating-from-stitching/