Apollo Federation: Connecting GraphQL Microservices

Apollo federation enables you to architect, build and connect multiple distributed GraphQL microservices. In a microservices world, you'll eventually run into the situation where you need to query distributed APIs. GraphQL microservices are no different. Whether they are created using AWS AppSync or Apollo Server, you have to query these distributed GraphQL microservices in your client application. If you had to connect to each one in your client application, you must authenticate against all these GraphQL APIs as they have different domains. Apollo Federation Gateway enables you to combine the distributed GraphQL subgraphs into one so that your client sends queries to one endpoint.

{Allen Azemia}

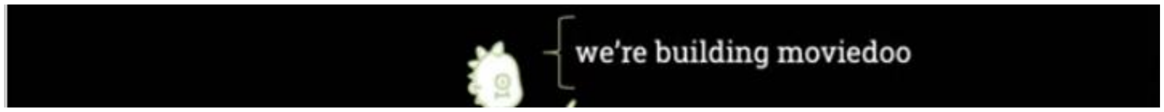
A Father, Developer, Lead Consultant @ Telstra Purple, Perth.

I am passionate about products.

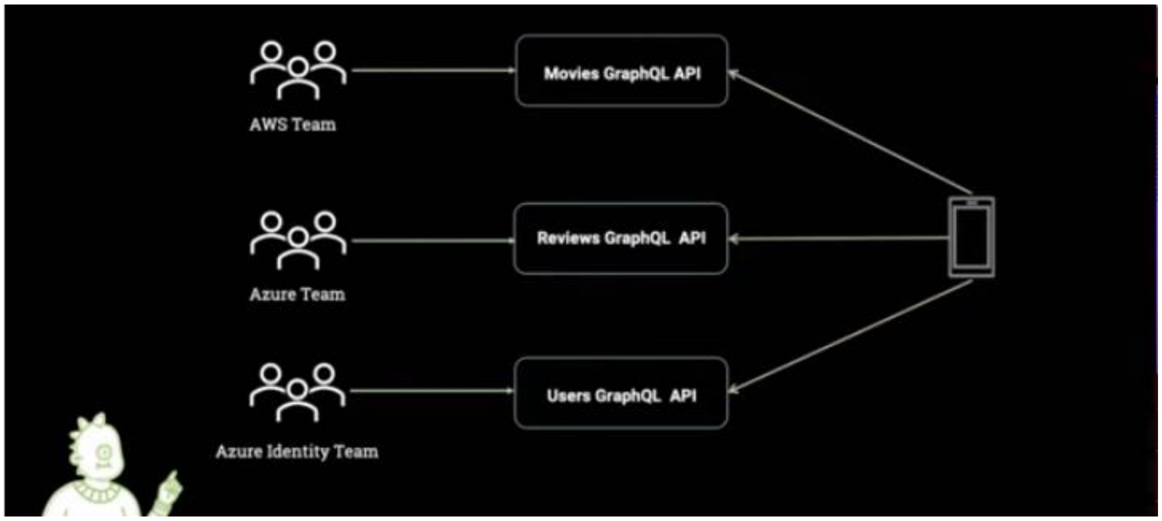
@allazemia

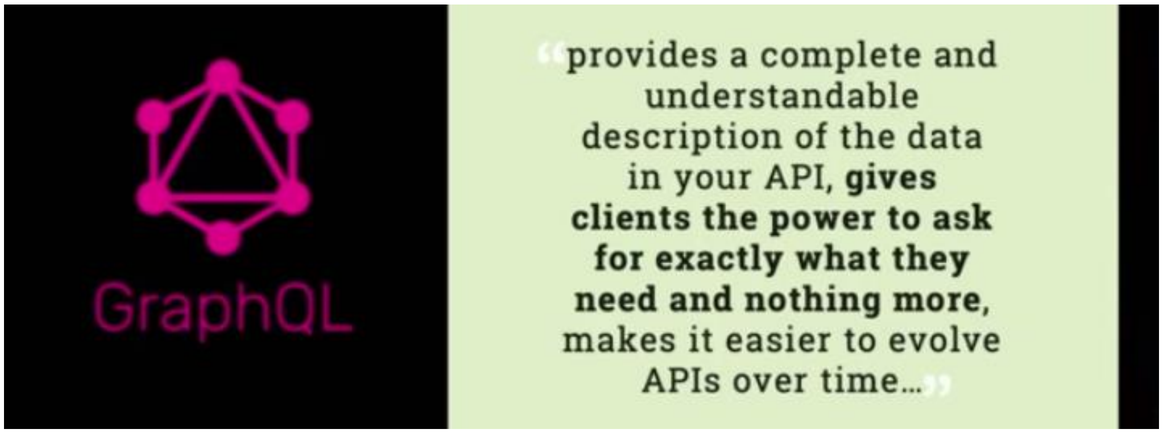
www.linkedin.com/in/allenazemia

allen.azemia@purple.tesltra.com



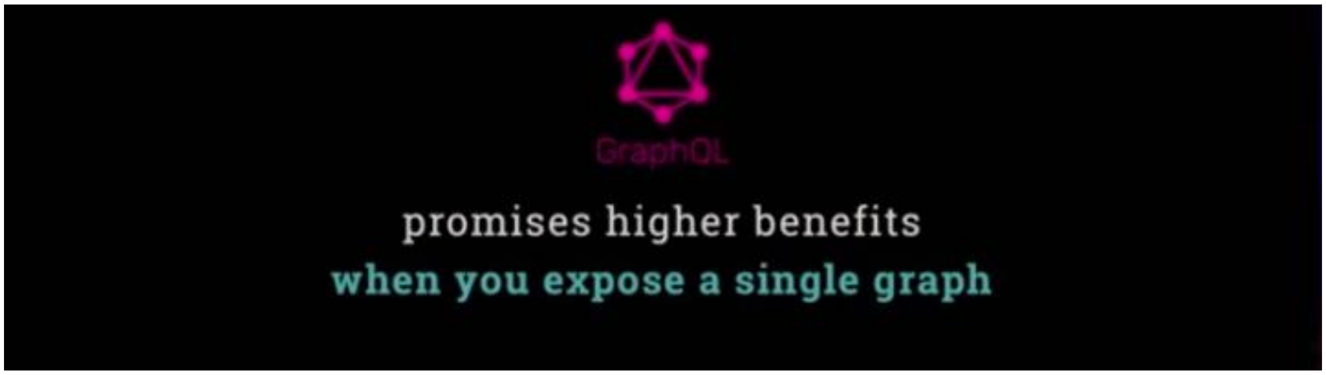
we're building moviedoo





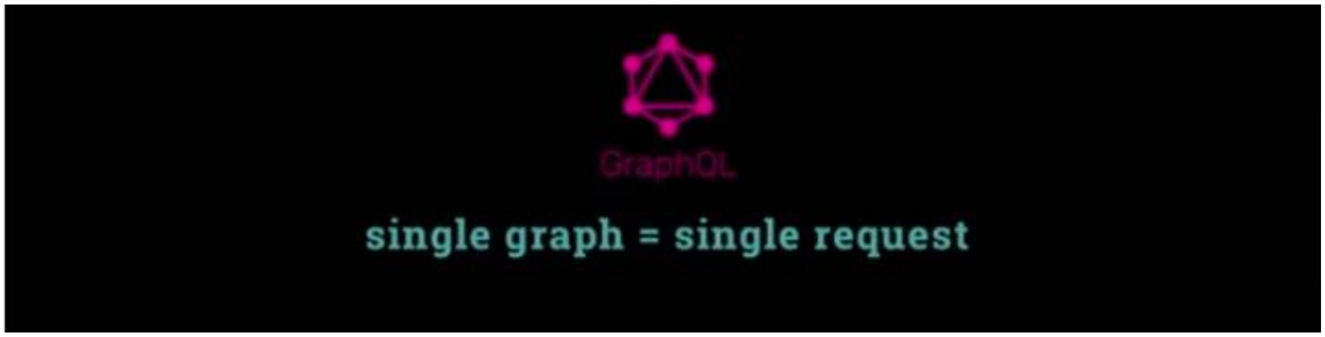
GraphQL

“provides a complete and understandable description of the data in your API, **gives clients the power to ask for exactly what they need and nothing more**, makes it easier to evolve APIs over time…”



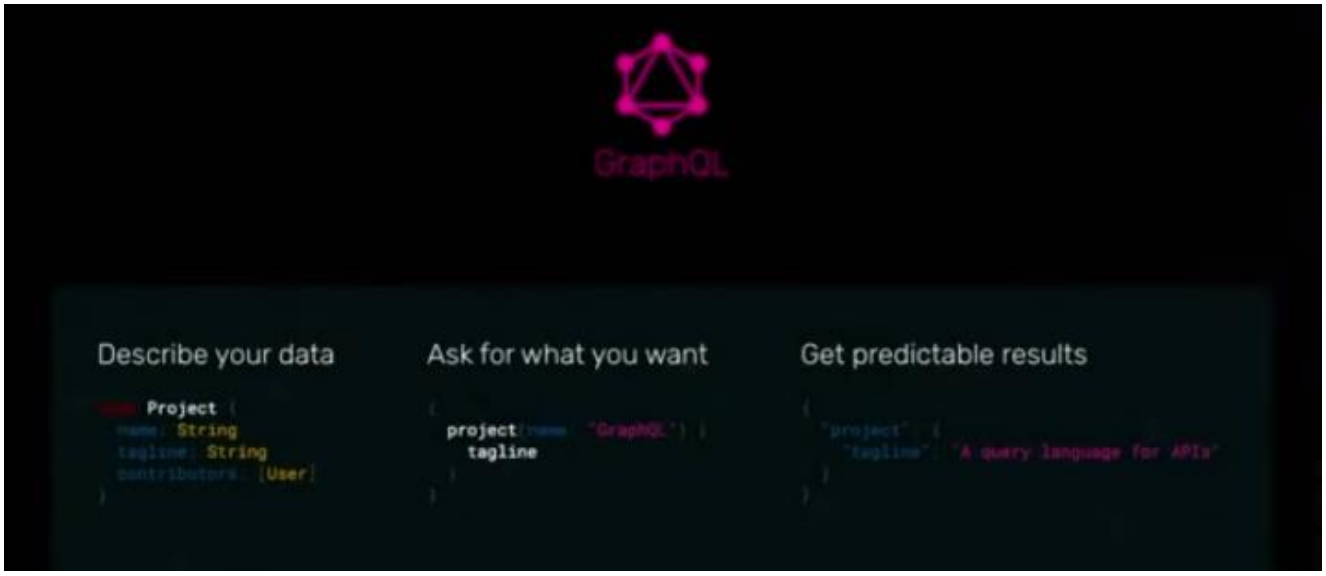
GraphQL

promises higher benefits
when you expose a single graph



GraphQL

single graph = single request



GraphQL

Describe your data

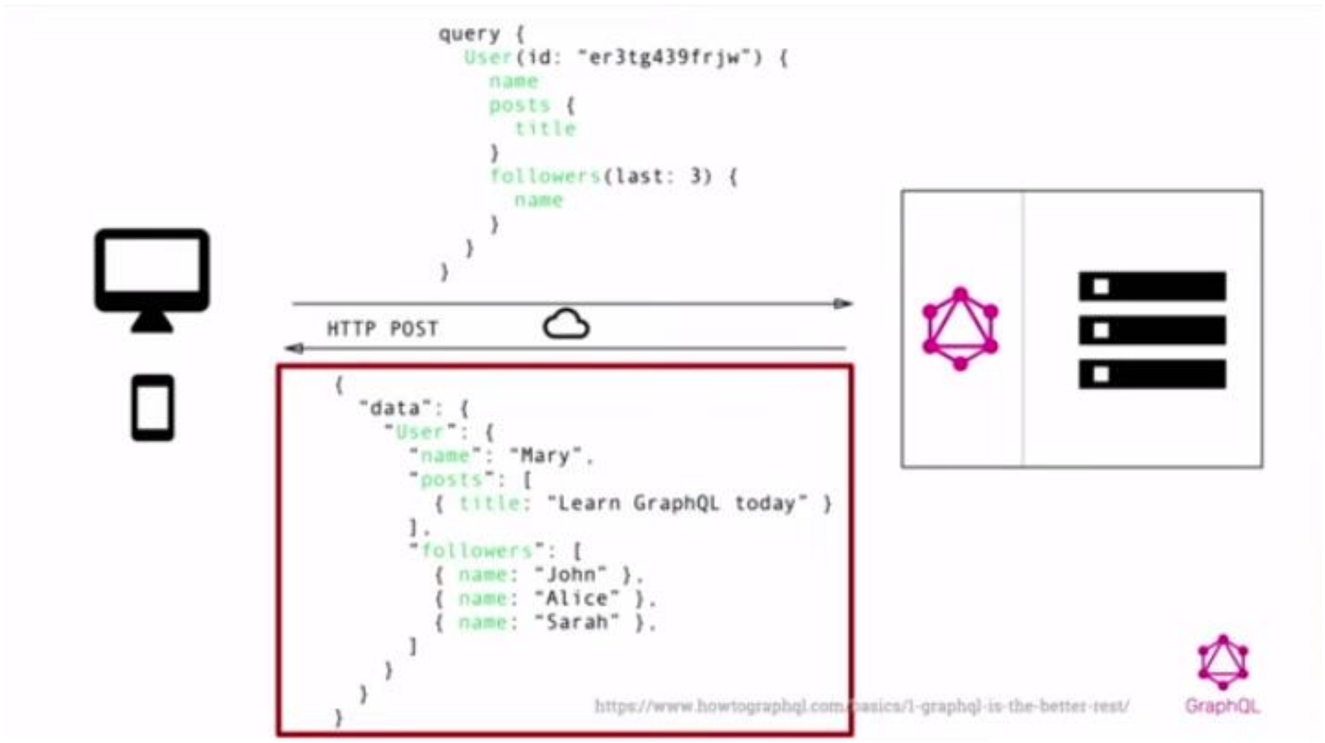
```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

Ask for what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

Get predictable results

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```



```
query {
  User(id: "er3tg439frjw") {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

```
{
  "data": {
    "User": {
      "name": "Mary",
      "posts": [
        { title: "Learn GraphQL today" }
      ],
      "followers": [
        { name: "John" },
        { name: "Alice" },
        { name: "Sarah" },
      ]
    }
  }
}
```

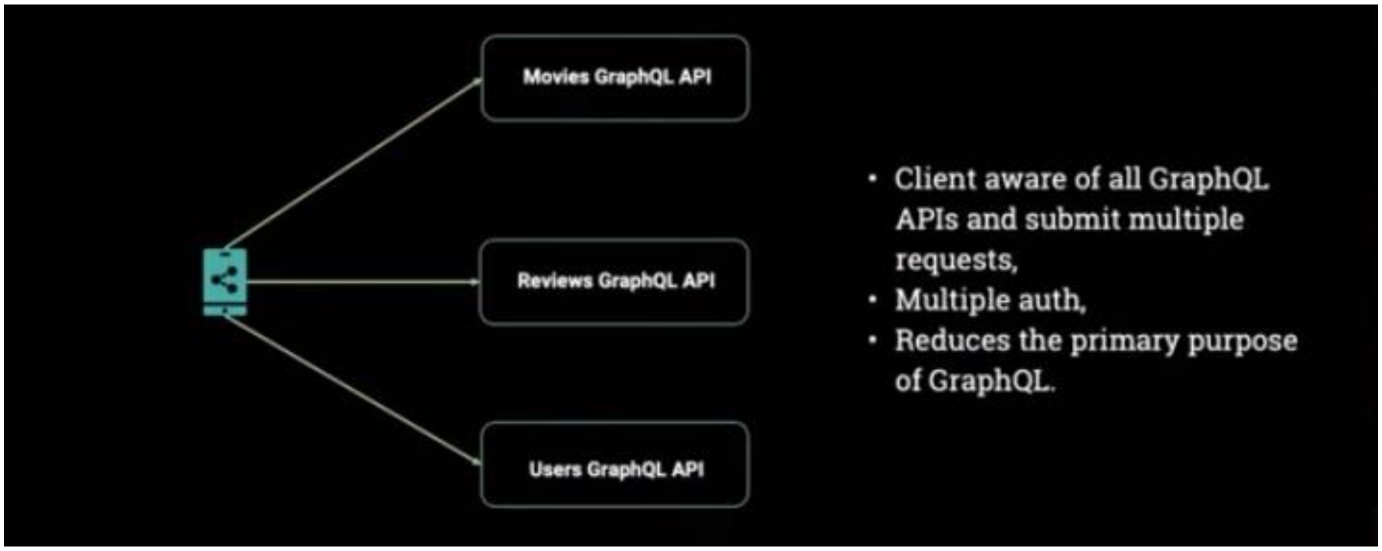
https://www.howtographql.com/basics/1-graphql-is-the-better-rest/
GraphQL

The problem

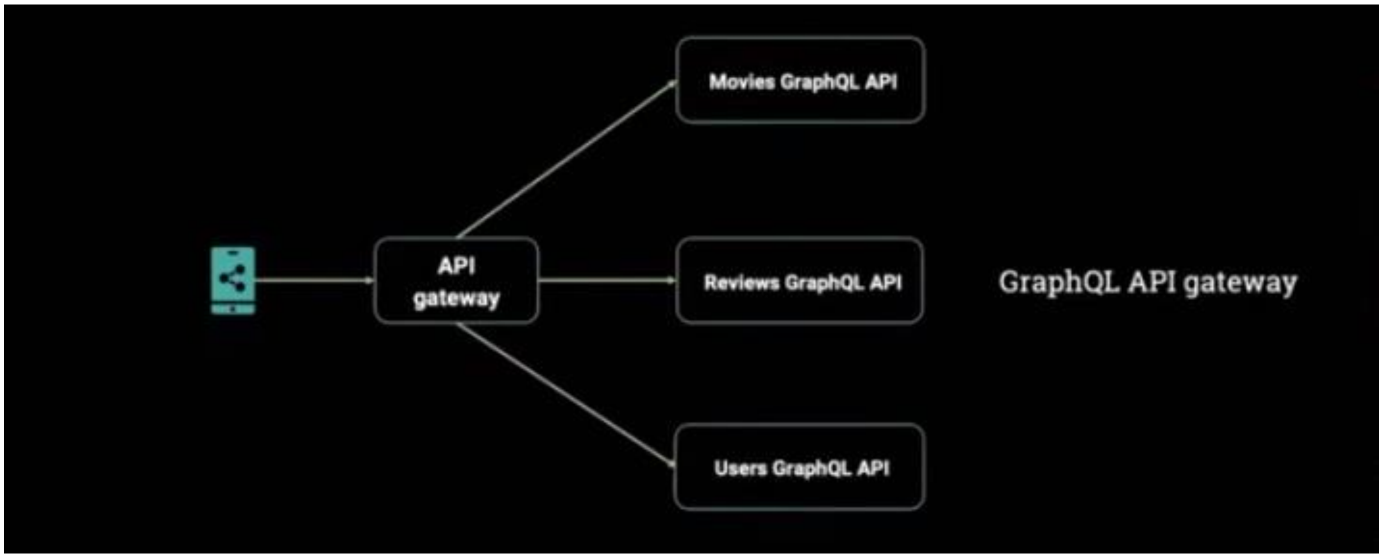
"MULTIPLE OVERLAPPING GRAPHS..."

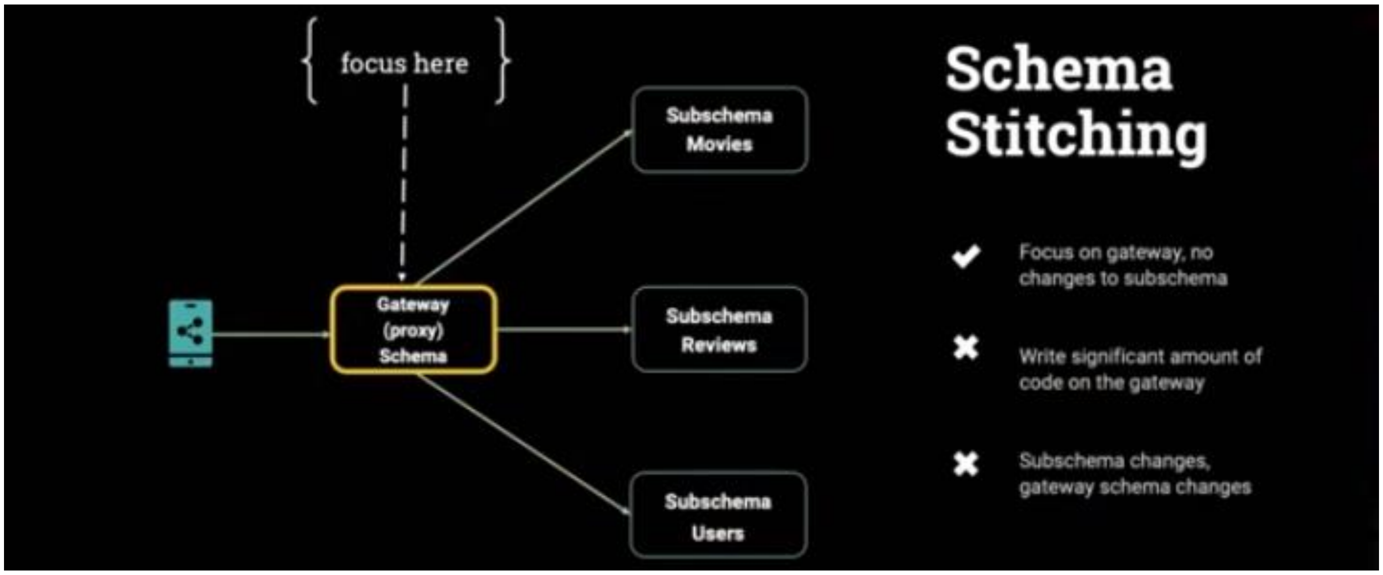
Apollo

...multiple development teams building independent GraphQL APIs.



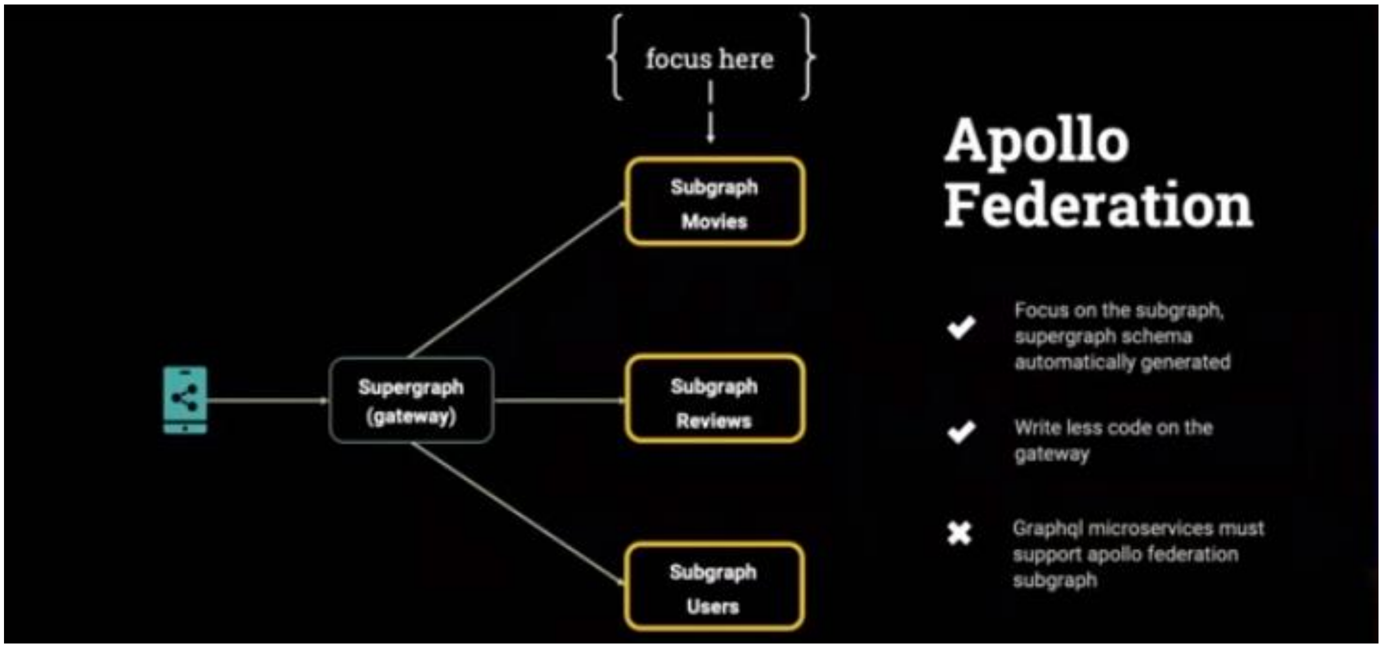
- Client aware of all GraphQL APIs and submit multiple requests,
- Multiple auth,
- Reduces the primary purpose of GraphQL.





Schema Stitching

- ✓ Focus on gateway, no changes to subschema
- ✖ Write significant amount of code on the gateway
- ✖ Subschema changes, gateway schema changes



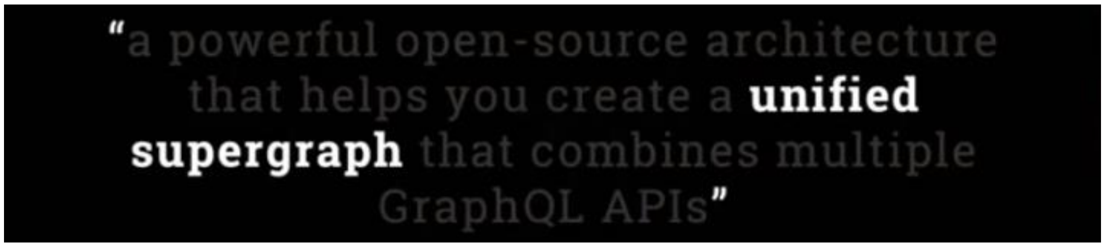
Apollo Federation

- ✓ Focus on the subgraph, supergraph schema automatically generated
- ✓ Write less code on the gateway
- ✖ Graphql microservices must support apollo federation subgraph

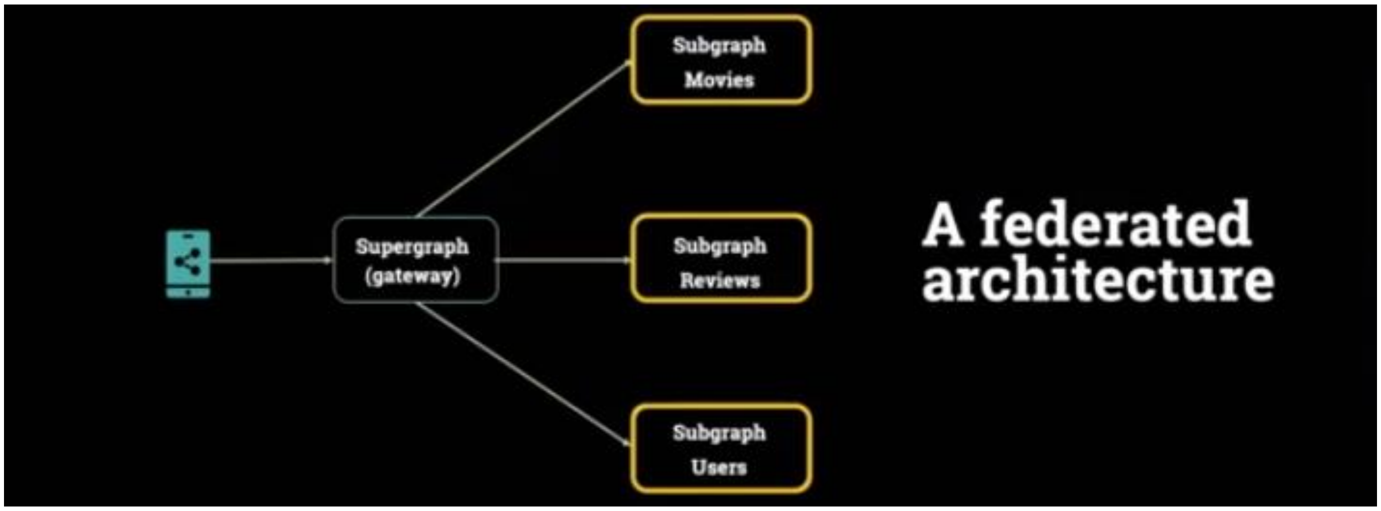
But your subgraph needs to understand Apollo federation and use some specific directives that Apollo federation provides and support so that your supergraph can interpret the subgraph schemas and create the supergraph for you.

Apollo Federation

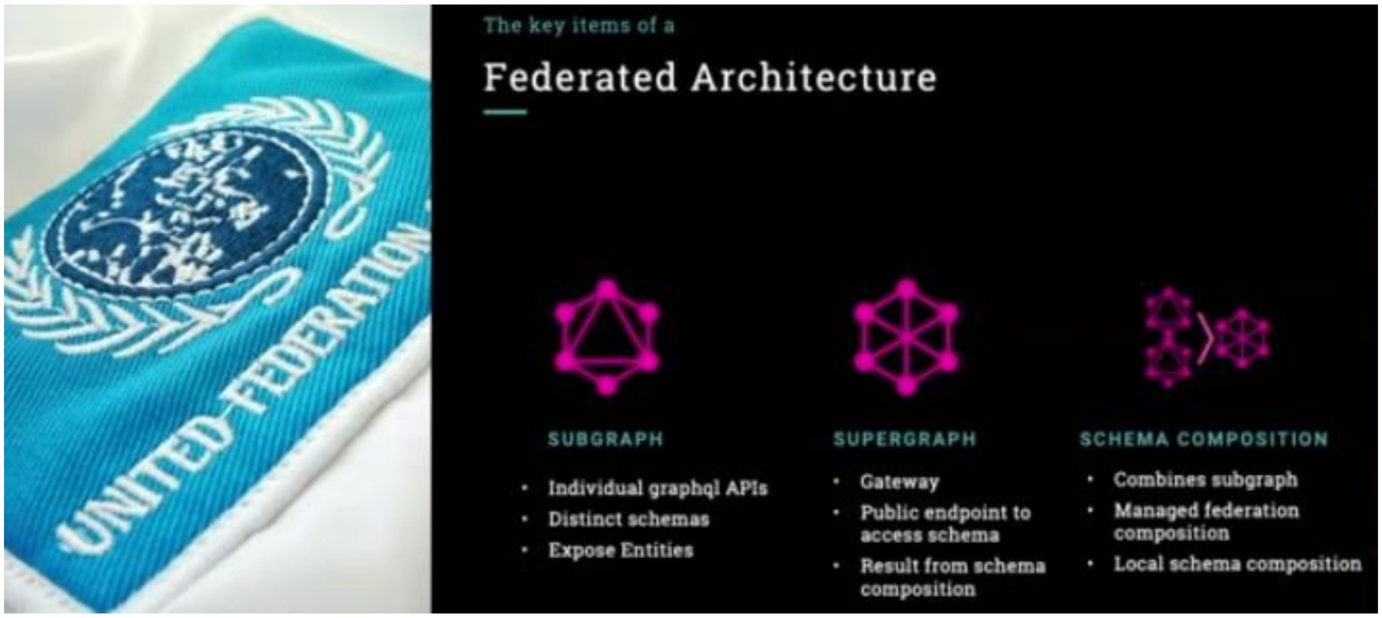
tell me more...



"a powerful open-source architecture that helps you create a **unified supergraph** that combines multiple GraphQL APIs"



A federated architecture



The key items of a

Federated Architecture

SUBGRAPH

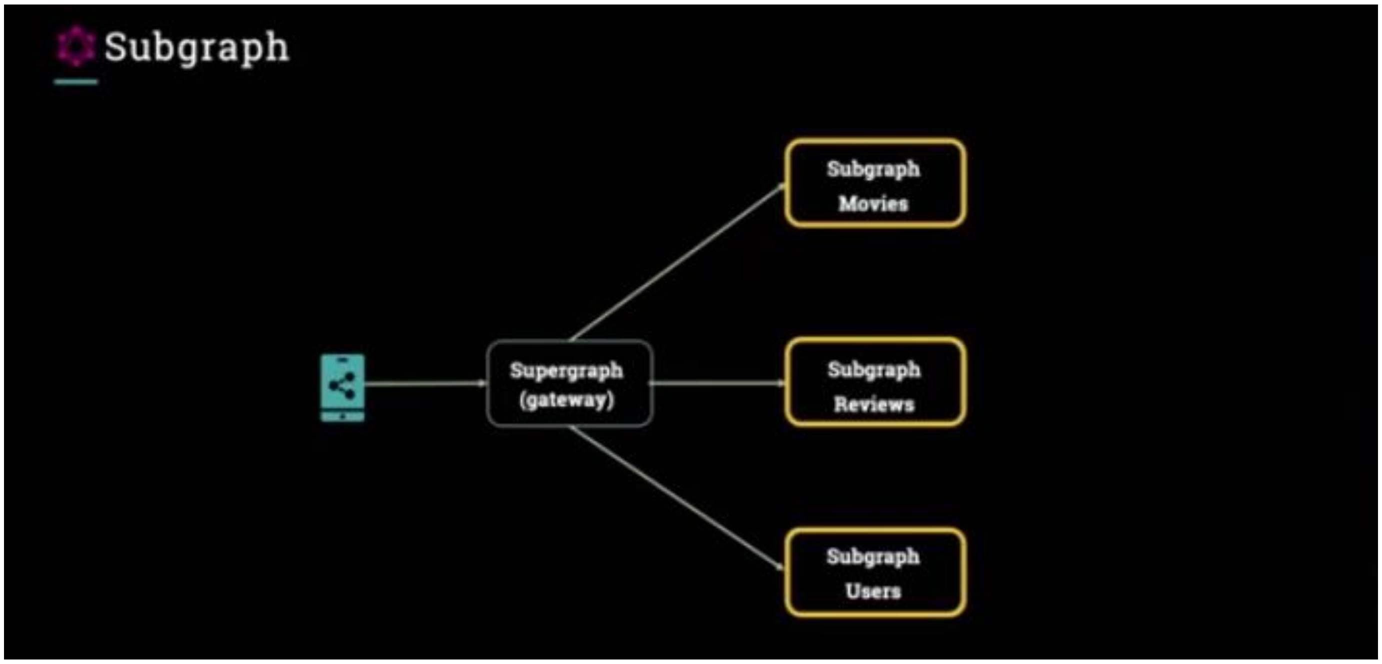
- Individual graphql APIs
- Distinct schemas
- Expose Entities

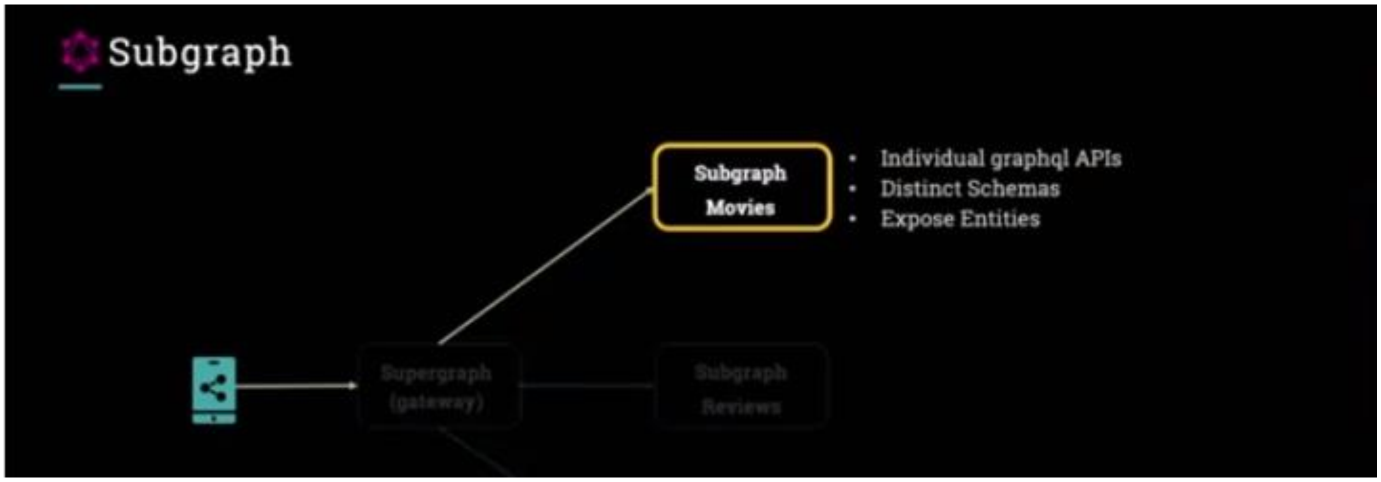
SUPERGRAPH

- Gateway
- Public endpoint to access schema
- Result from schema composition

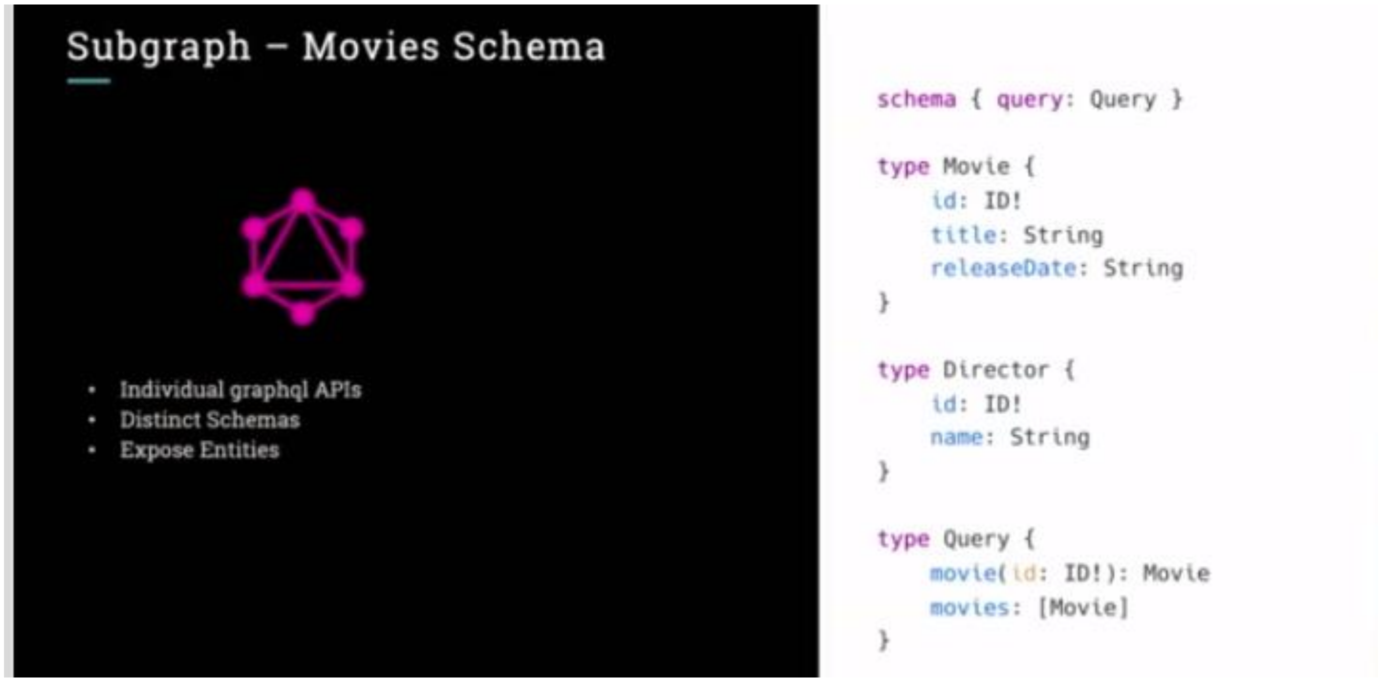
SCHEMA COMPOSITION

- Combines subgraph
- Managed federation composition
- Local schema composition





- Individual graphql APIs
- Distinct Schemas
- Expose Entities

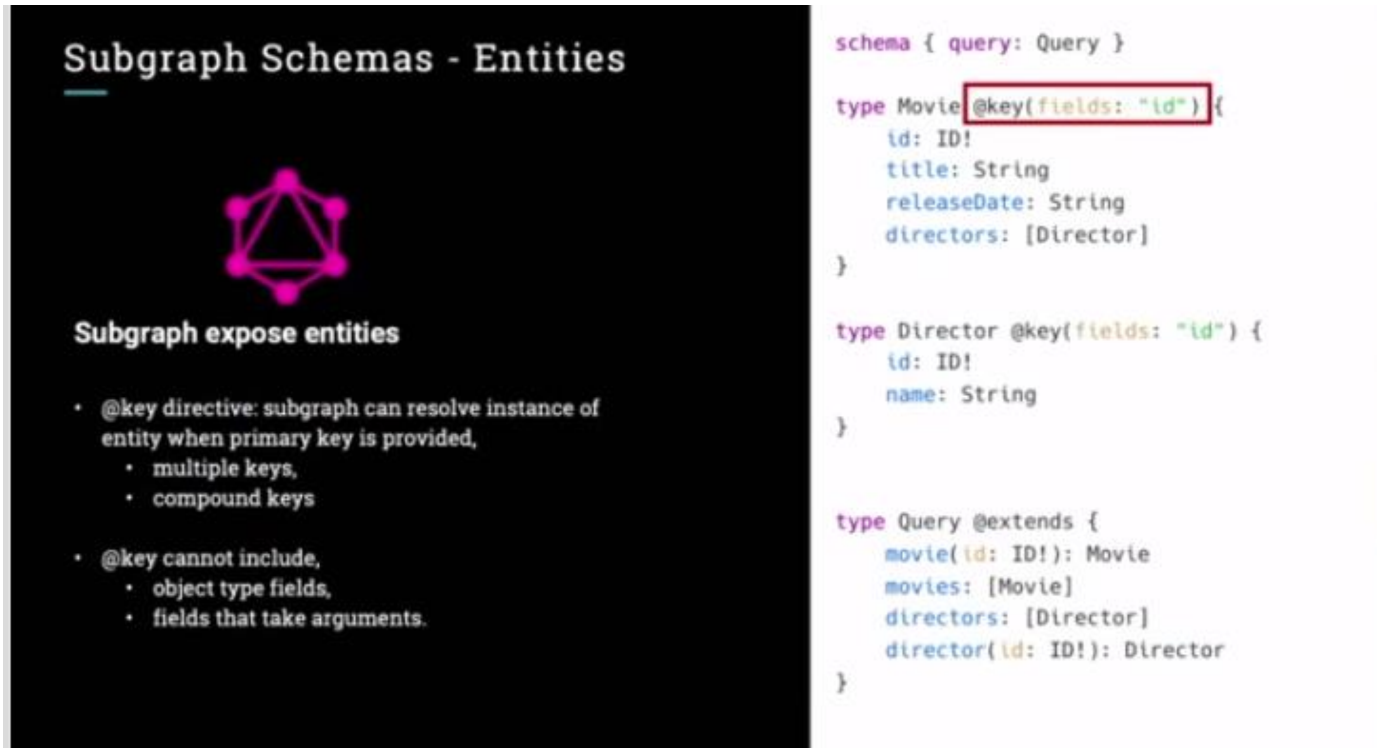


Subgraph – Movies Schema

- Individual graphql APIs
- Distinct Schemas
- Expose Entities

```
schema { query: Query }

type Movie {
    id: ID!
    title: String
    releaseDate: String
}

type Director {
    id: ID!
    name: String
}

type Query {
    movie(id: ID!): Movie
    movies: [Movie]
}
```

Apollo federation might be able to read the above BUT it won't be able to interpret it because of no schema directives

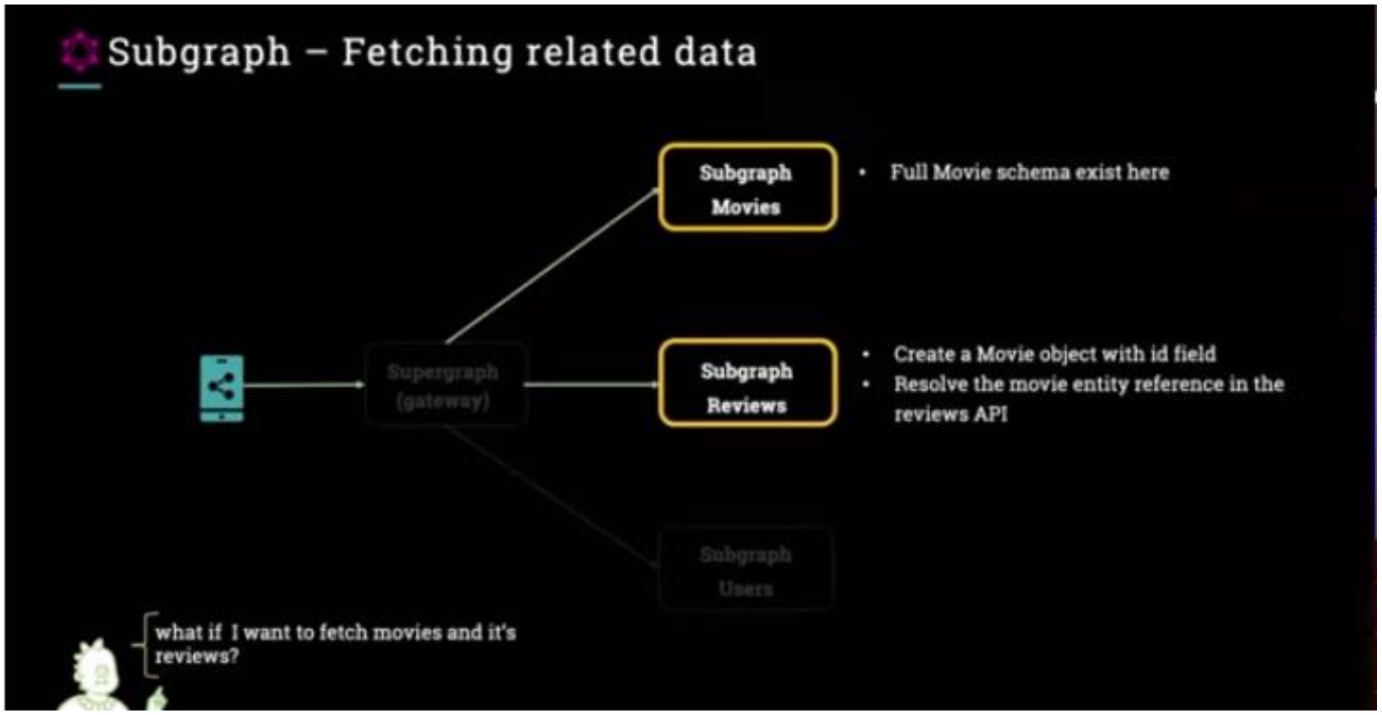


Subgraph Schemas - Entities

Subgraph expose entities

- @key directive: subgraph can resolve instance of entity when primary key is provided,
  - multiple keys,
  - compound keys
- @key cannot include,
  - object type fields,
  - fields that take arguments.

```
schema { query: Query }

type Movie @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String
    directors: [Director]
}

type Director @key(fields: "id") {
    id: ID!
    name: String
}


type Query @extends {
    movie(id: ID!): Movie
    movies: [Movie]
    directors: [Director]
    director(id: ID!): Director
}
```

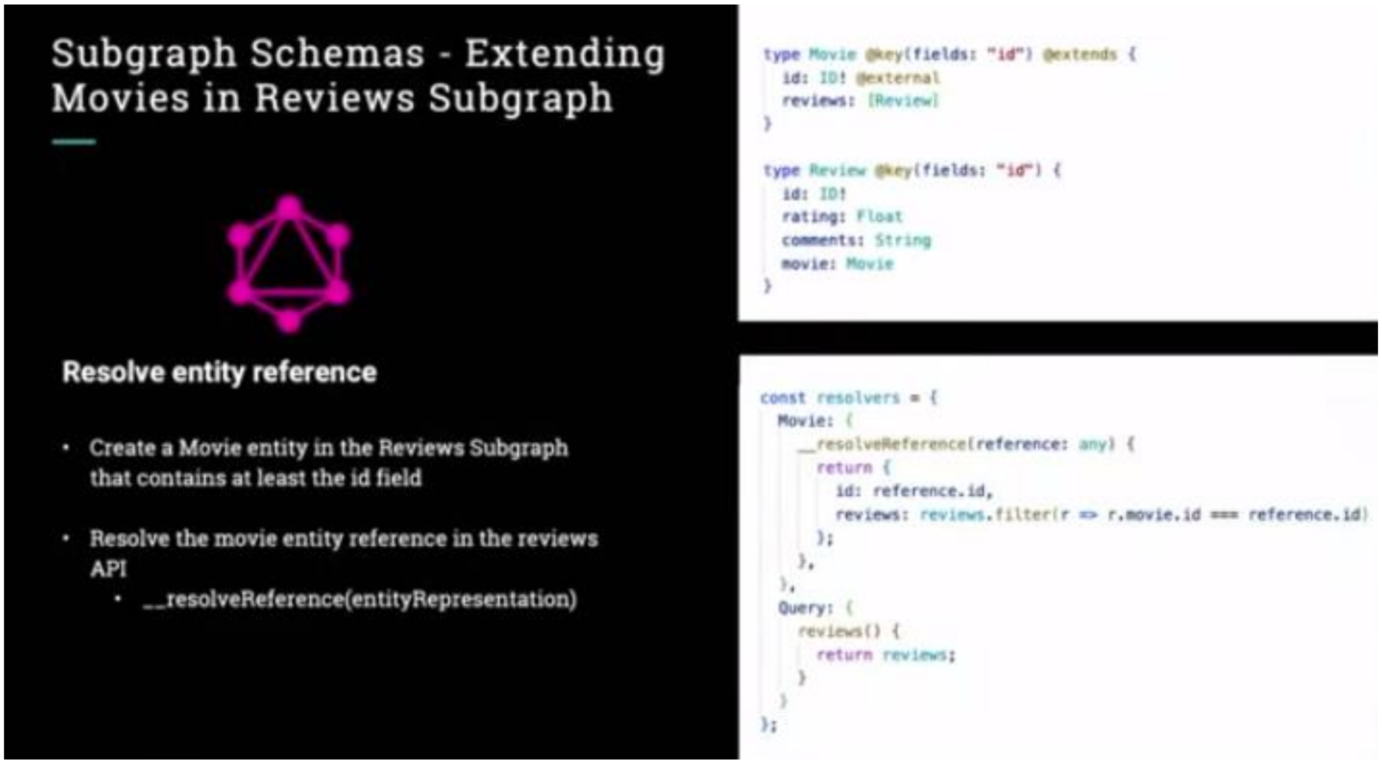


Subgraph – Fetching related data

- Full Movie schema exist here

- Create a Movie object with id field
- Resolve the movie entity reference in the reviews API

what if I want to fetch movies and it's reviews?



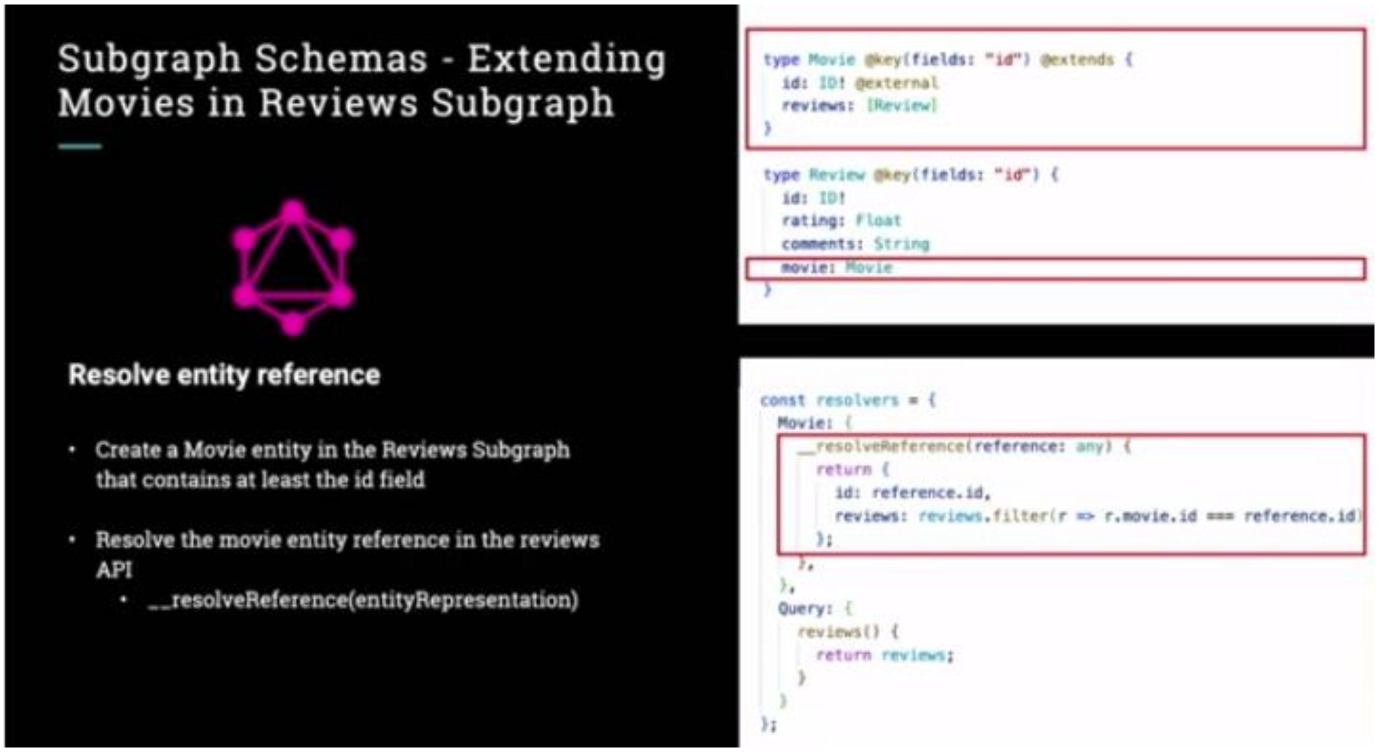
Subgraph Schemas - Extending Movies in Reviews Subgraph

Resolve entity reference

- Create a Movie entity in the Reviews Subgraph that contains at least the id field
- Resolve the movie entity reference in the reviews API
  - __resolveReference(entityRepresentation)

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```

```
const resolvers = {
  Movie: {
    __resolveReference(reference: any) {
      return {
        id: reference.id,
        reviews: reviews.filter(r => r.movie.id === reference.id)
      };
    },
  },
  Query: {
    reviews() {
      return reviews;
    }
  }
};
```



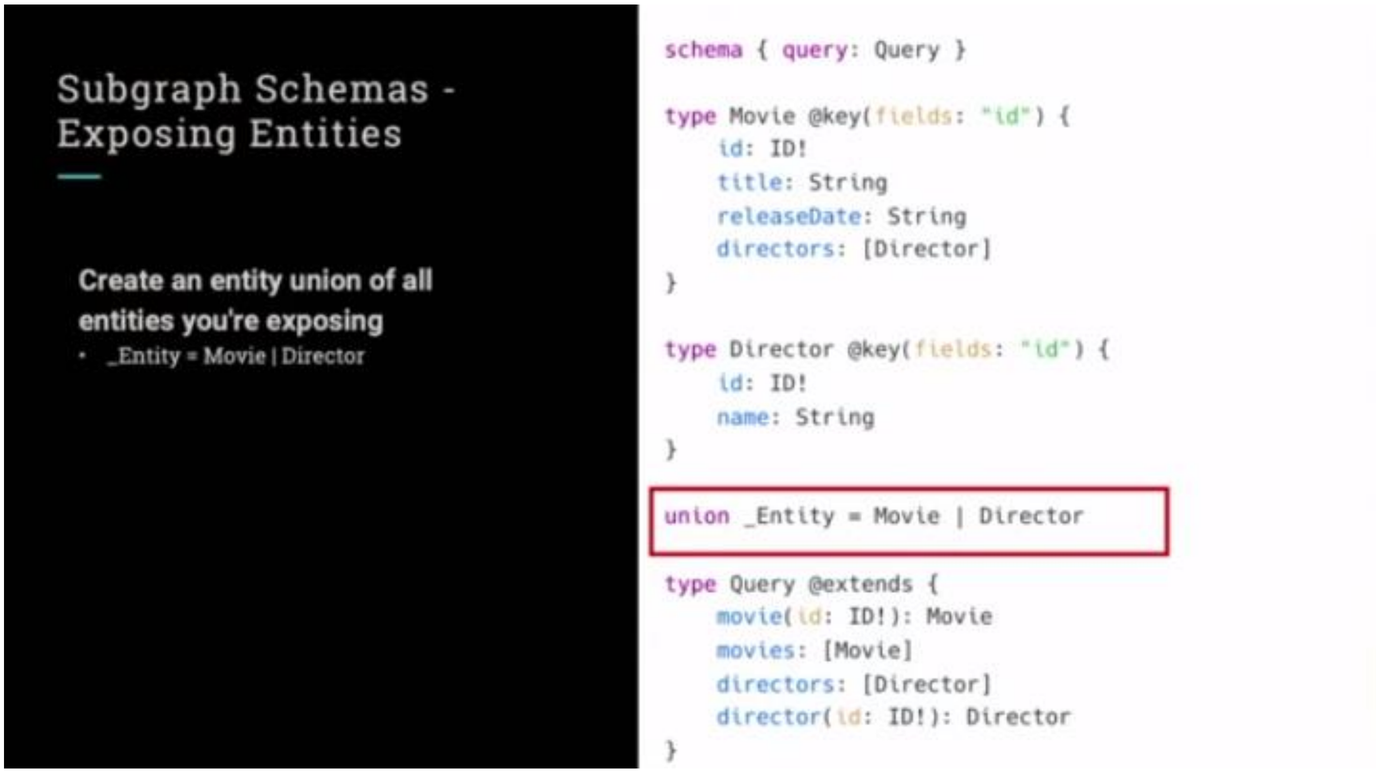
Subgraph Schemas - Extending Movies in Reviews Subgraph

Resolve entity reference

- Create a Movie entity in the Reviews Subgraph that contains at least the id field
- Resolve the movie entity reference in the reviews API
  - __resolveReference(entityRepresentation)

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```

```
const resolvers = {
  Movie: {
    __resolveReference(reference: any) {
      return {
        id: reference.id,
        reviews: reviews.filter(r => r.movie.id === reference.id)
      };
    },
  },
  Query: {
    reviews() {
      return reviews;
    }
  }
};
```

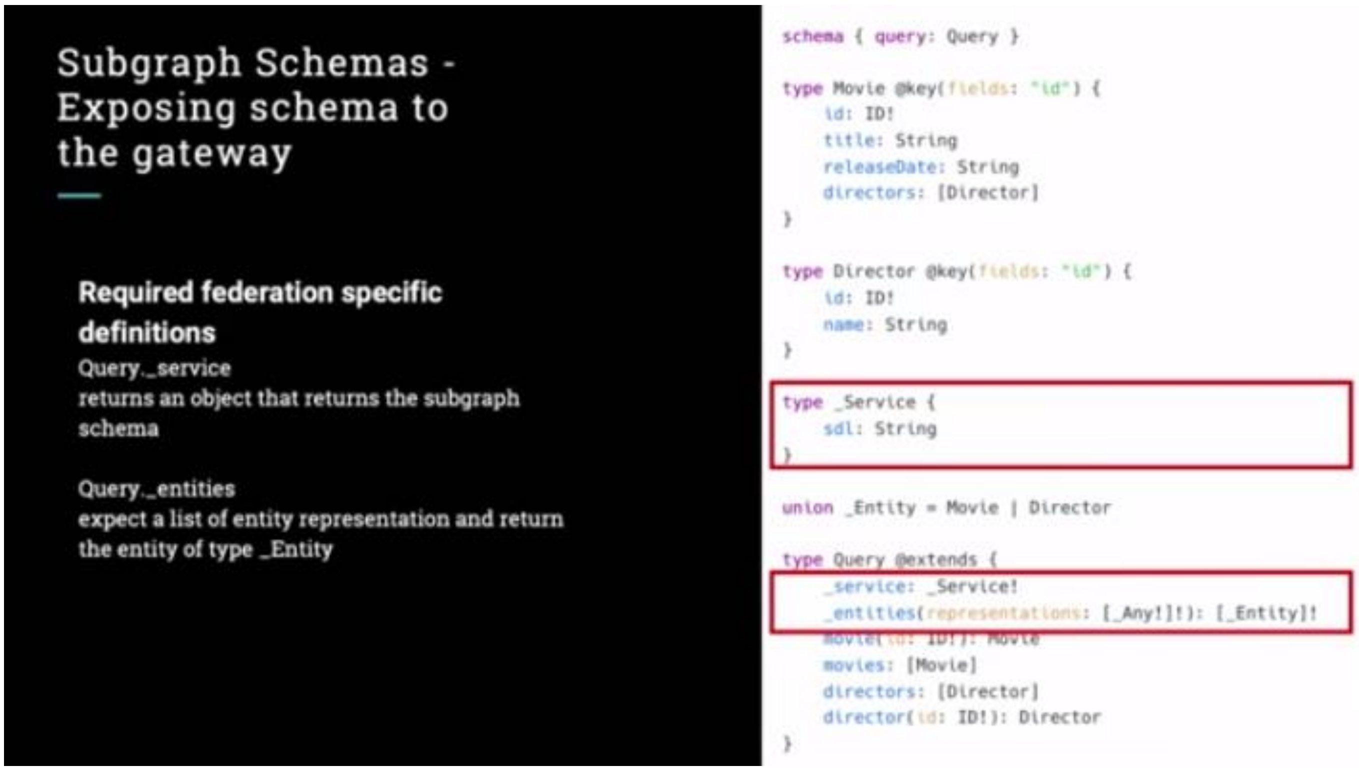


Subgraph Schemas - Exposing Entities

Create an entity union of all entities you're exposing

- _Entity = Movie | Director

```
schema { query: Query }

type Movie @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String
    directors: [Director]
}

type Director @key(fields: "id") {
    id: ID!
    name: String
}

union _Entity = Movie | Director

type Query @extends {
    movie(id: ID!): Movie
    movies: [Movie]
    directors: [Director]
    director(id: ID!): Director
}
```



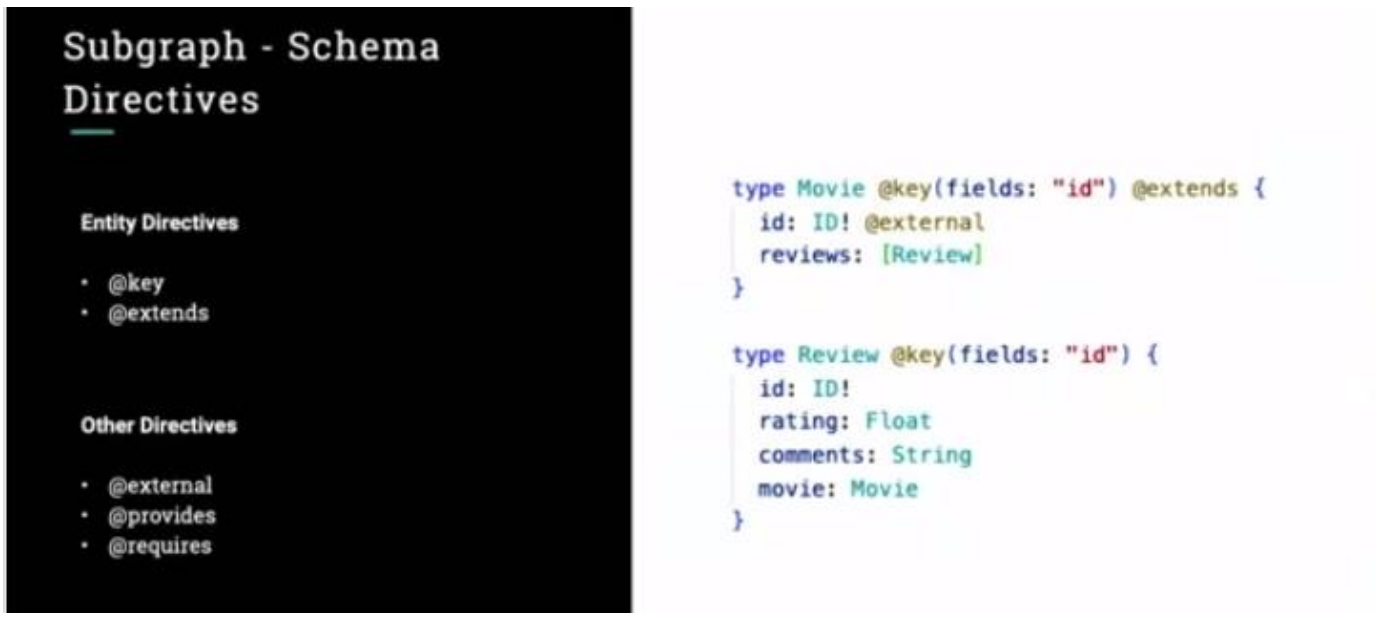
Subgraph Schemas - Exposing schema to the gateway

Required federation specific definitions

Query._service
returns an object that returns the subgraph schema

Query._entities
expect a list of entity representation and return the entity of type _Entity

```
schema { query: Query }

type Movie @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String
    directors: [Director]
}

type Director @key(fields: "id") {
    id: ID!
    name: String
}

type _Service {
    sdl: String
}

union _Entity = Movie | Director

type Query @extends {
    _service: _Service!
    _entities(representations: [_Any!]!): [_Entity]!
    movie(id: ID!): Movie
    movies: [Movie]
    directors: [Director]
    director(id: ID!): Director
}
```



Subgraph - Schema Directives

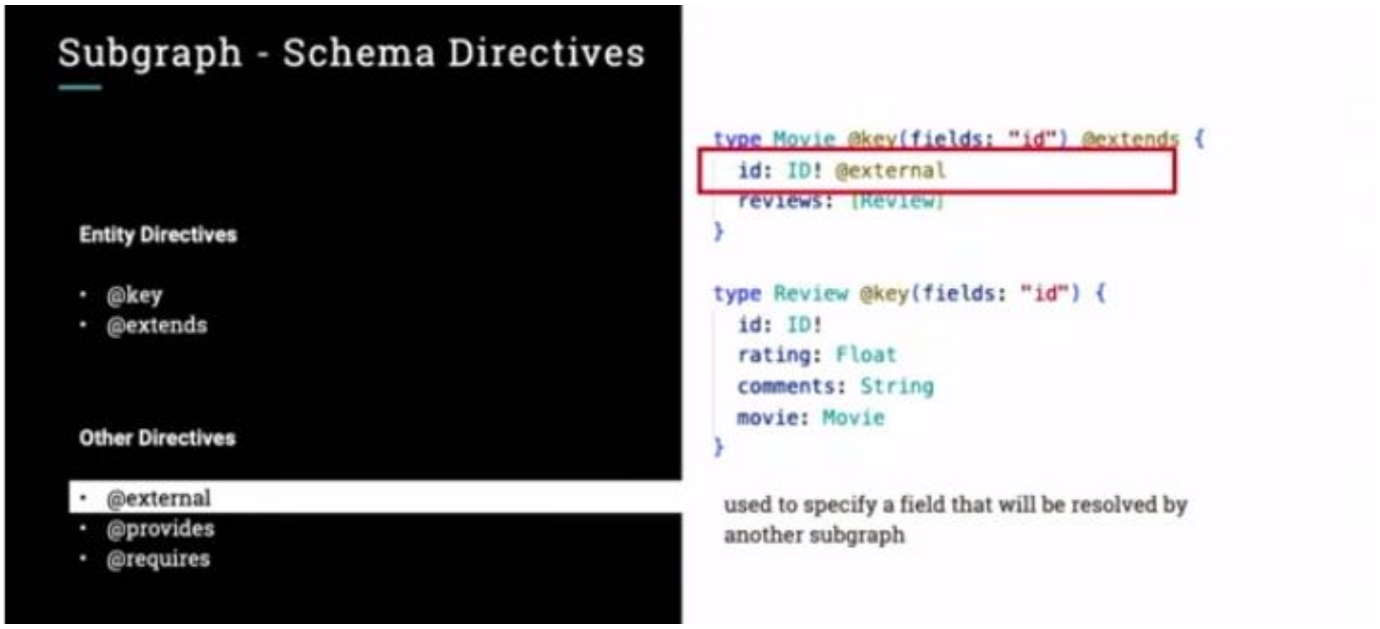
Entity Directives

- @key
- @extends

Other Directives

- @external
- @provides
- @requires

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```



Subgraph - Schema Directives

Entity Directives

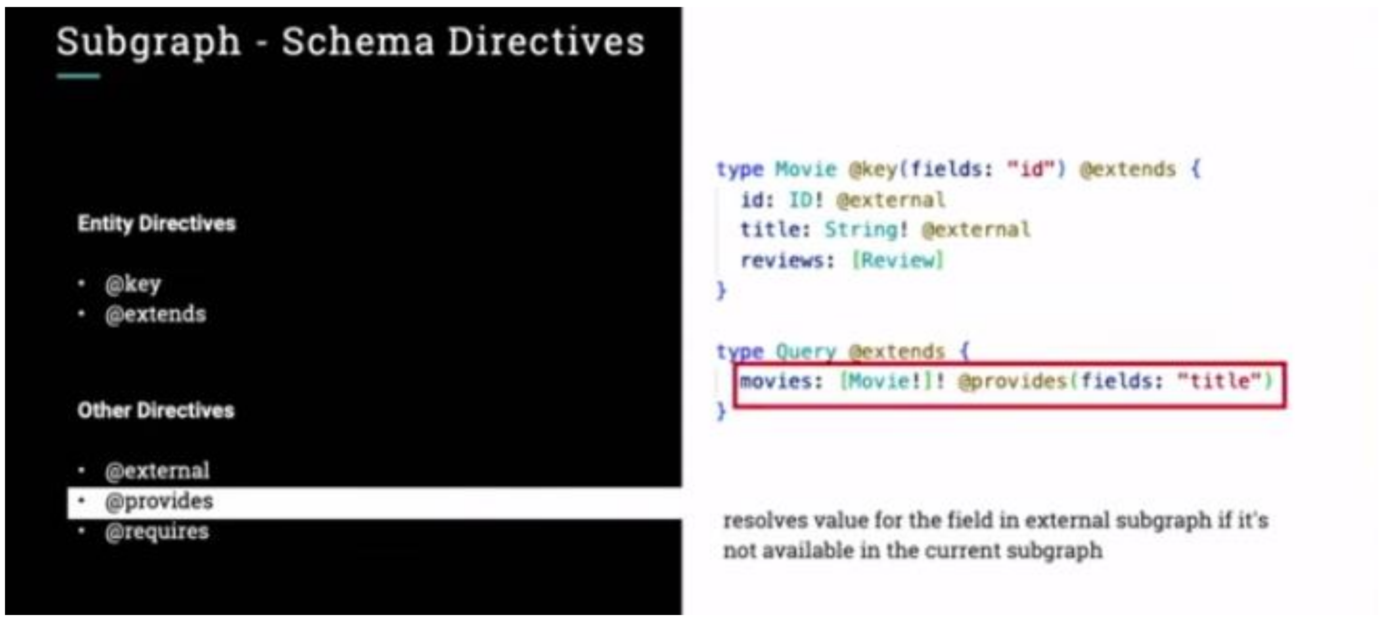
- @key
- @extends

Other Directives

- @external
- @provides
- @requires

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```

used to specify a field that will be resolved by another subgraph



Subgraph - Schema Directives

Entity Directives

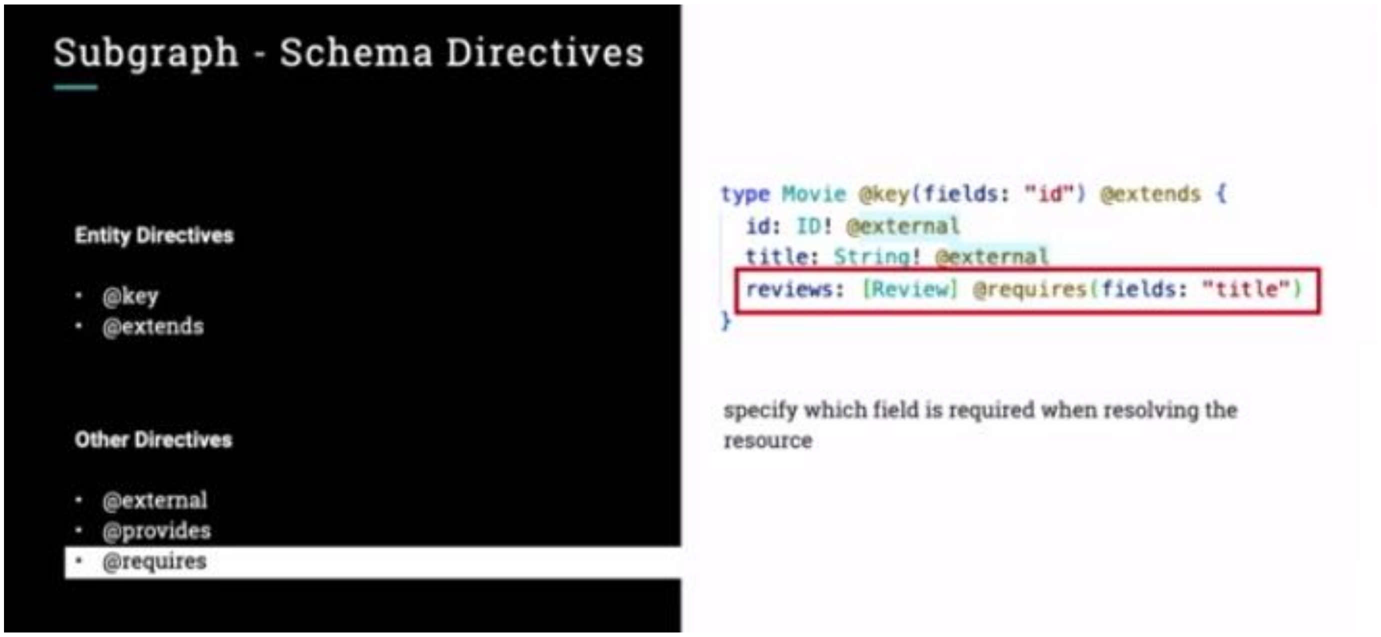
- @key
- @extends

Other Directives

- @external
- @provides
- @requires

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  title: String! @external
  reviews: [Review]
}

type Query @extends {
  movies: [Movie!]! @provides(fields: "title")
}
```

resolves value for the field in external subgraph if it's not available in the current subgraph



Subgraph - Schema Directives

Entity Directives

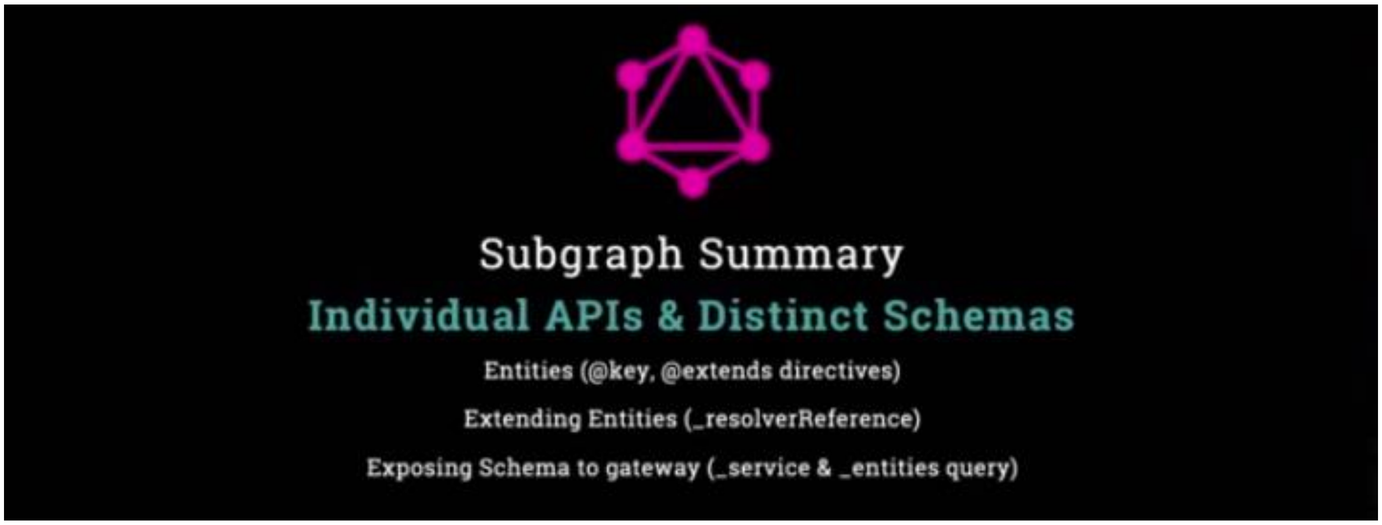
- @key
- @extends

Other Directives

- @external
- @provides
- @requires

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  title: String! @external
  reviews: [Review] @requires(fields: "title")
}
```

specify which field is required when resolving the resource



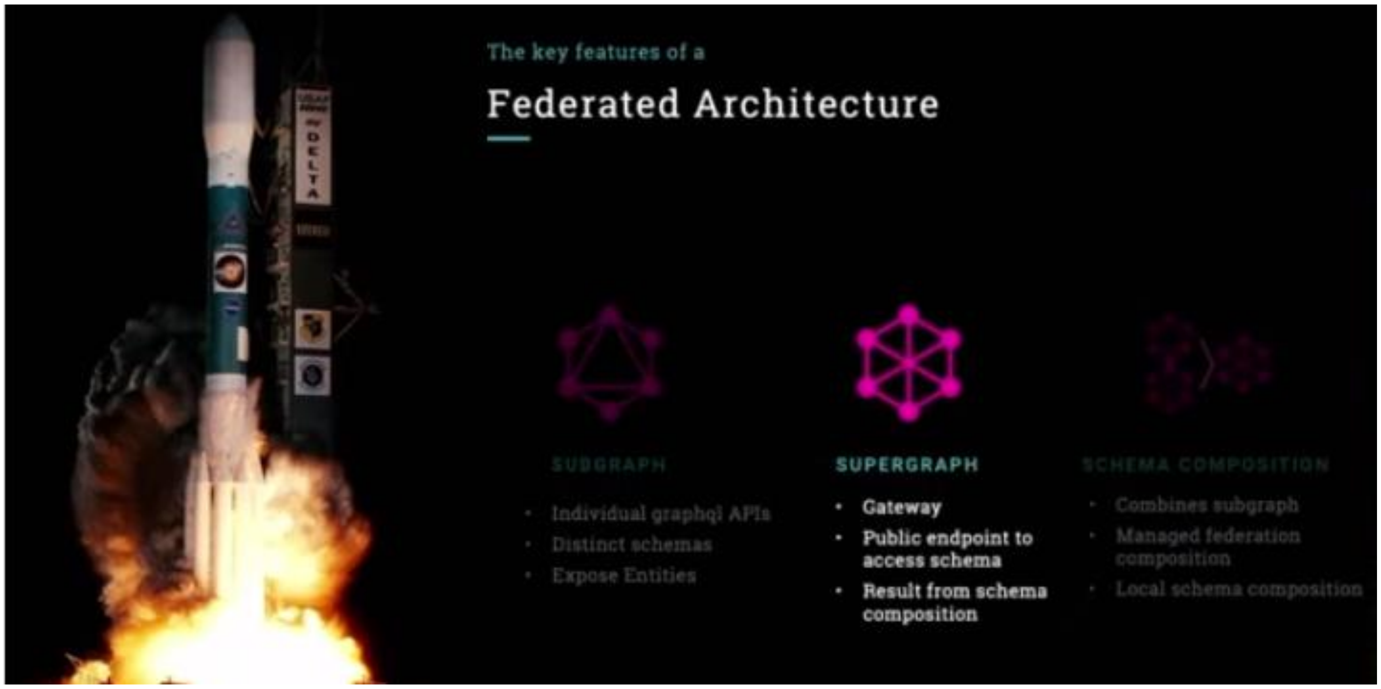
Subgraph Summary

Individual APIs & Distinct Schemas

Entities (@key, @extends directives)

Extending Entities (_resolverReference)

Exposing Schema to gateway (_service & _entities query)



The key features of a

Federated Architecture

SUBGRAPH

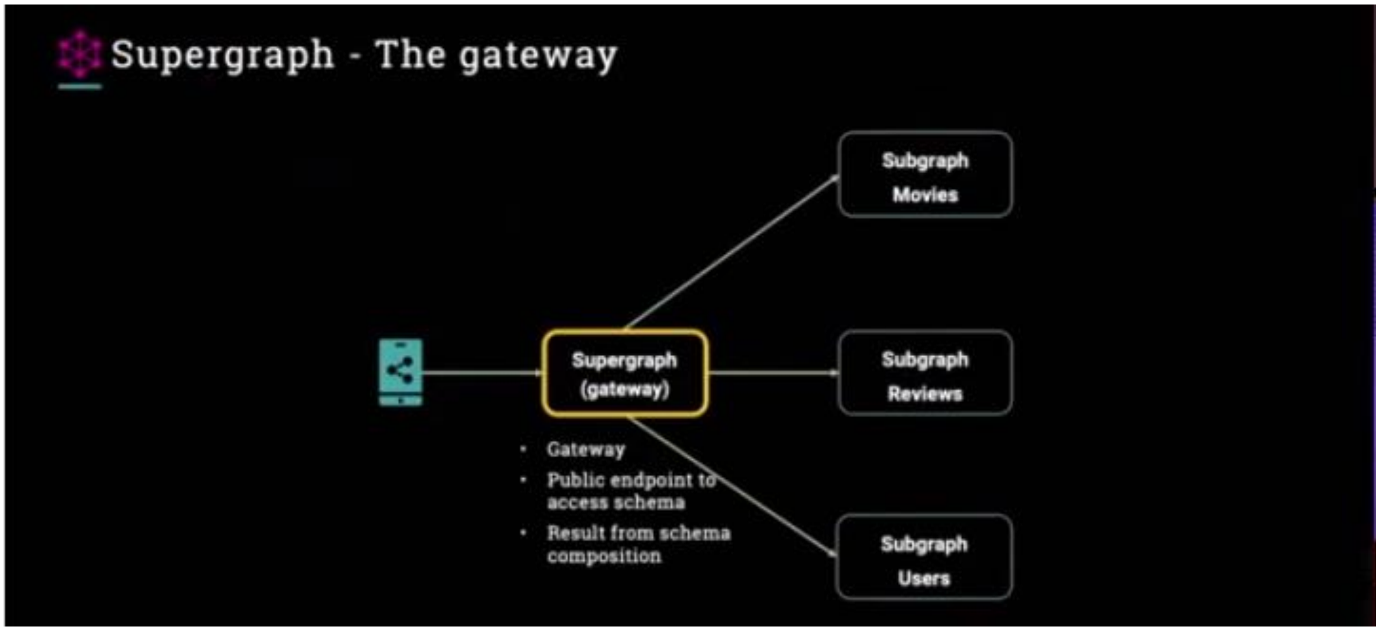
- Individual graphql APIs
- Distinct schemas
- Expose Entities

SUPERGRAPH

- Gateway
- Public endpoint to access schema
- Result from schema composition

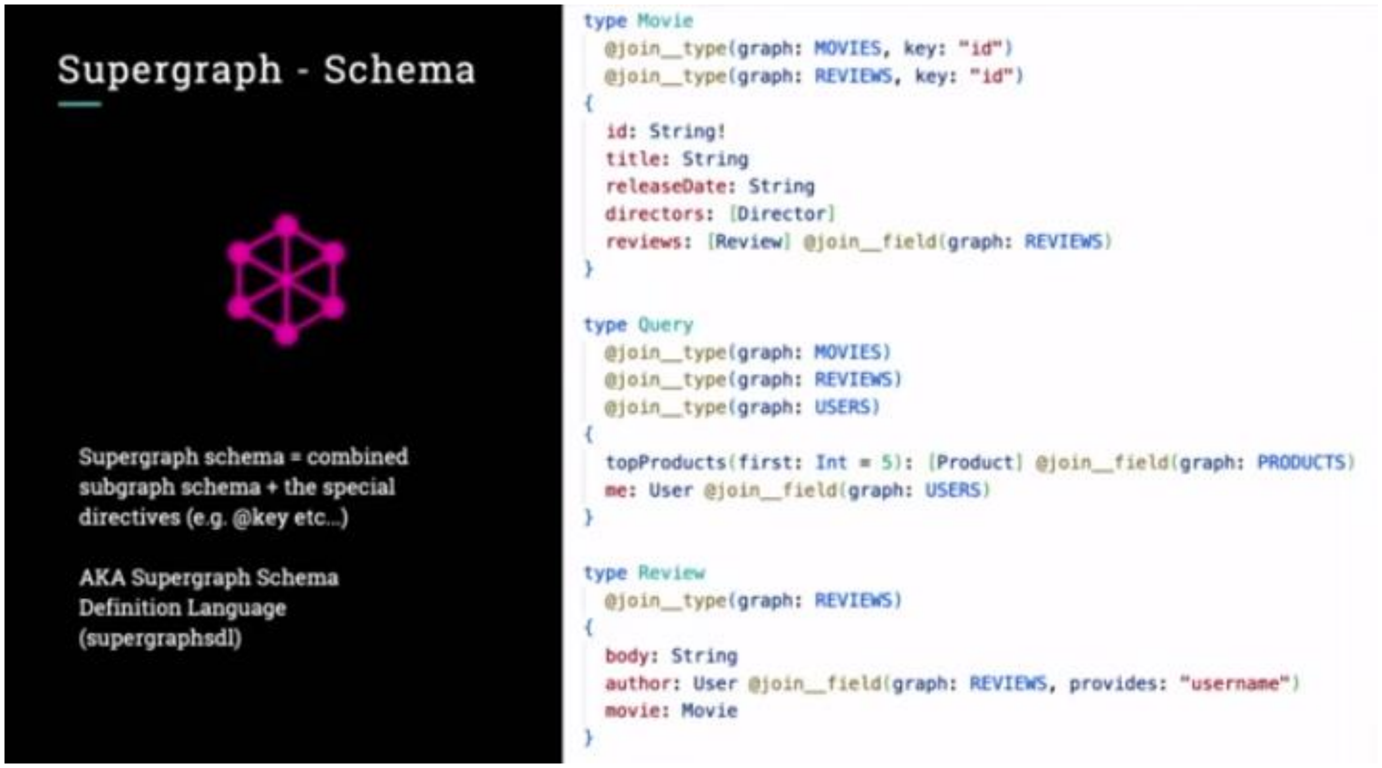
SCHEMA COMPOSITION

- Combines subgraph
- Managed federation composition
- Local schema composition



Supergraph - The gateway

- Gateway
- Public endpoint to access schema
- Result from schema composition

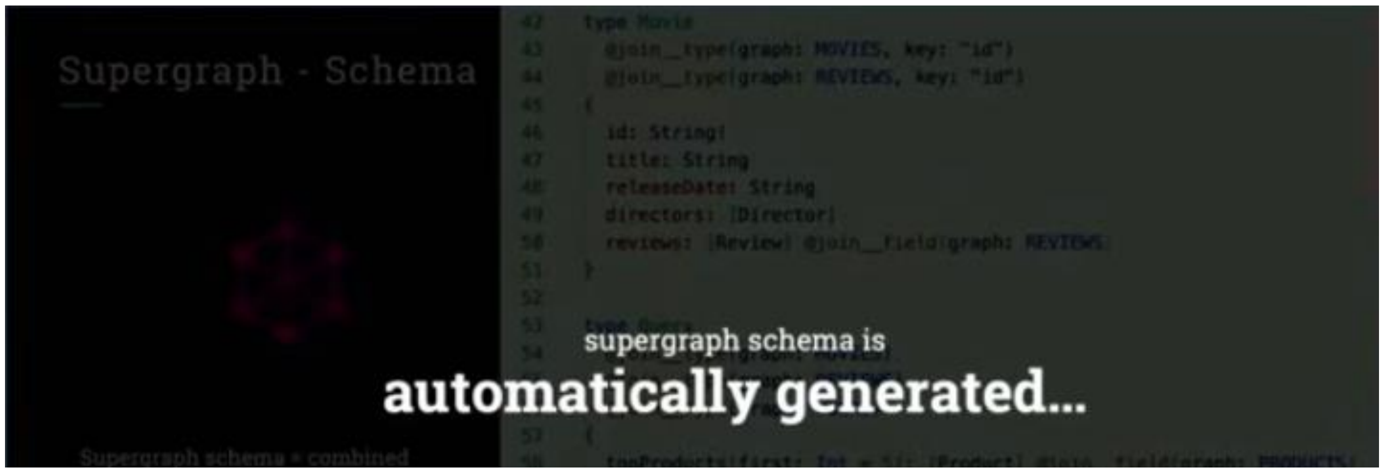


Supergraph - Schema

Supergraph schema = combined subgraph schema + the special directives (e.g. @key etc...)

AKA Supergraph Schema Definition Language (supergraphsdl)

```
type Movie
  @join__type(graph: MOVIES, key: "id")
  @join__type(graph: REVIEWS, key: "id")
{
  id: String!
  title: String
  releaseDate: String
  directors: [Director]
  reviews: [Review] @join__field(graph: REVIEWS)
}

type Query
  @join__type(graph: MOVIES)
  @join__type(graph: REVIEWS)
  @join__type(graph: USERS)
{
  topProducts(first: Int = 5): [Product] @join__field(graph: PRODUCTS)
  me: User @join__field(graph: USERS)
}

type Review
  @join__type(graph: REVIEWS)
{
  body: String
  author: User @join__field(graph: REVIEWS, provides: "username")
  movie: Movie
}
```



Supergraph - Schema

```
type Movie
  @join__type(graph: MOVIES, key: "id")
  @join__type(graph: REVIEWS, key: "id")
{
  id: String!
  title: String
  releaseDate: String
  directors: [Director]
  reviews: [Review] @join__field(graph: REVIEWS)
}
```

Supergraph schema = combined

supergraph schema is
automatically generated...

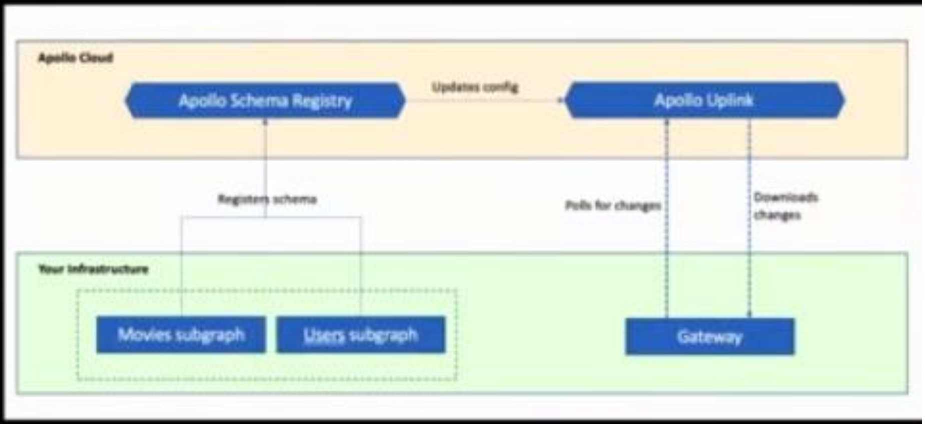
Supergraph – Schema Composition

Apollo Studio

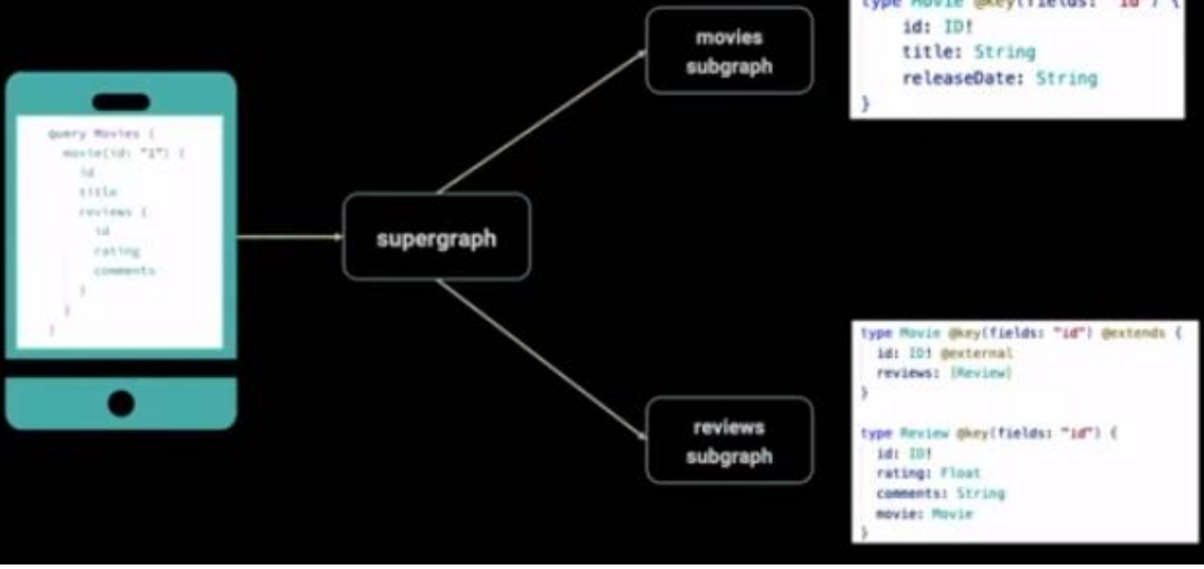
+

Rover CLI

- Subgraph schema registered in schema registry
- Supergraph + config stored in Apollo Uplink
- Gateway polls Uplink



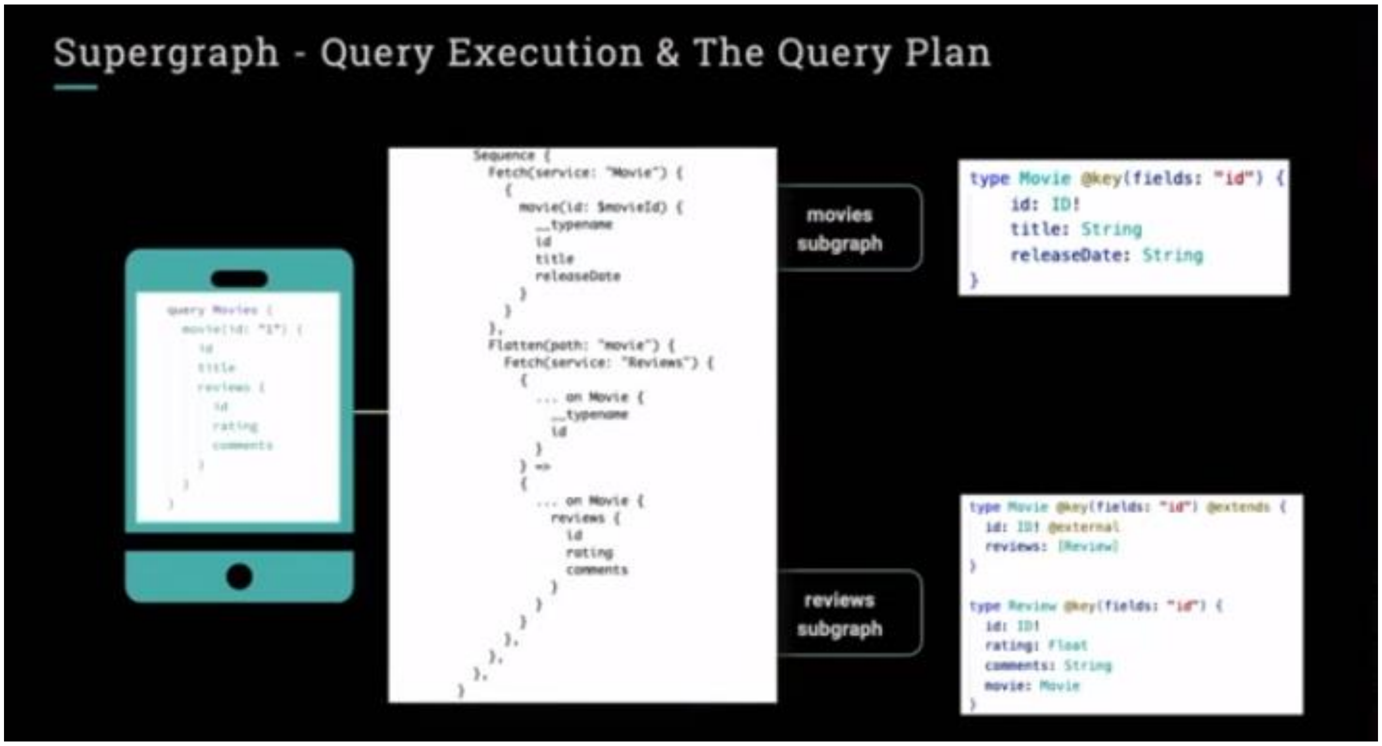
Supergraph - Query Execution & The Query Plan



```
query Movies {
  movie(id: "1") {
    id
    title
    reviews {
      id
      rating
      comments
    }
  }
}
```

```
type Movie @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String
}
```

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```



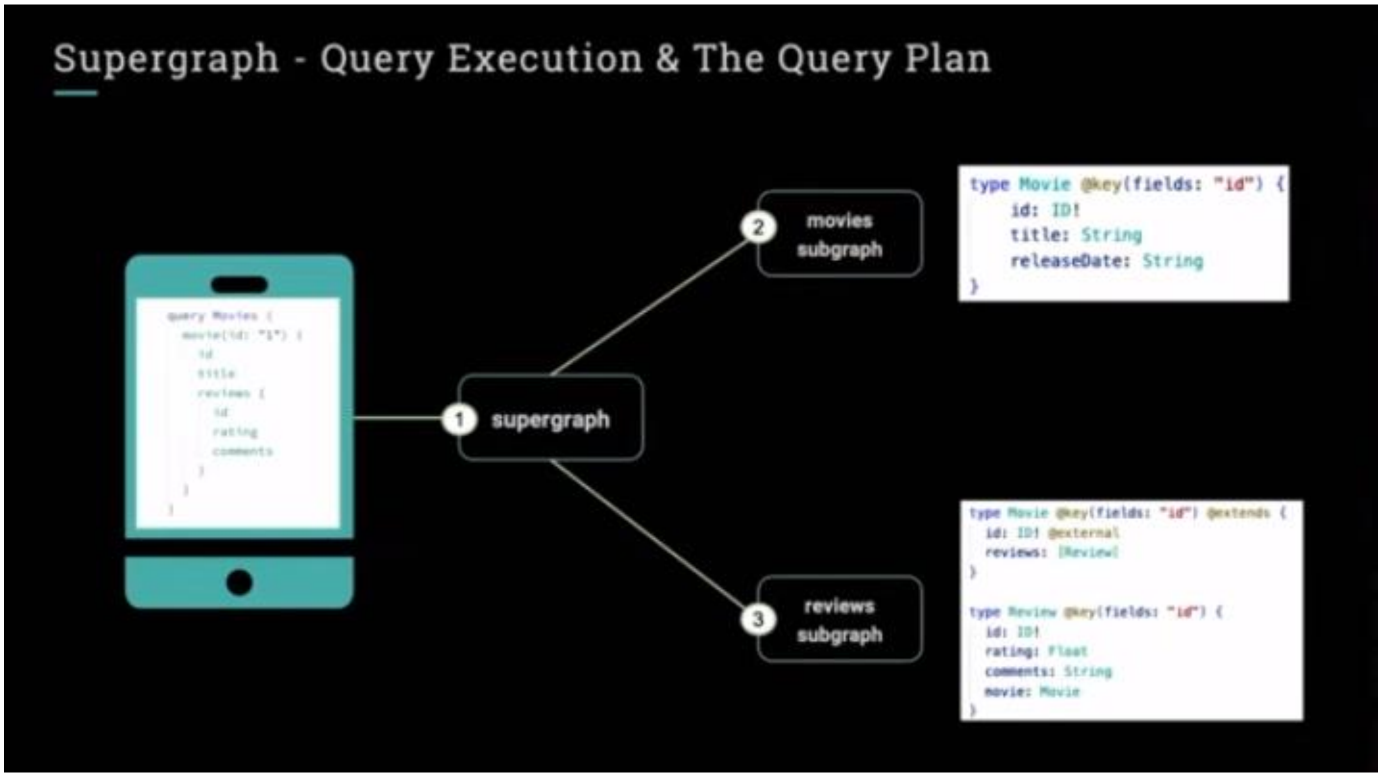
Supergraph - Query Execution & The Query Plan

```
query Movies {
  movie(id: "1") {
    id
    title
    reviews {
      id
      rating
      comments
    }
  }
}
```

```
Sequence {
  Fetch(service: "Movie") {
    {
      movie(id: $movieId) {
        __typename
        id
        title
        releaseDate
      }
    }
  },
  Flatten(path: "movie") {
    Fetch(service: "Reviews") {
      {
        ... on Movie {
          __typename
          id
        }
      } =>
      {
        ... on Movie {
          reviews {
            id
            rating
            comments
          }
        }
      }
    },
  },
 },
}
```

```
type Movie @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String
}
```

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```

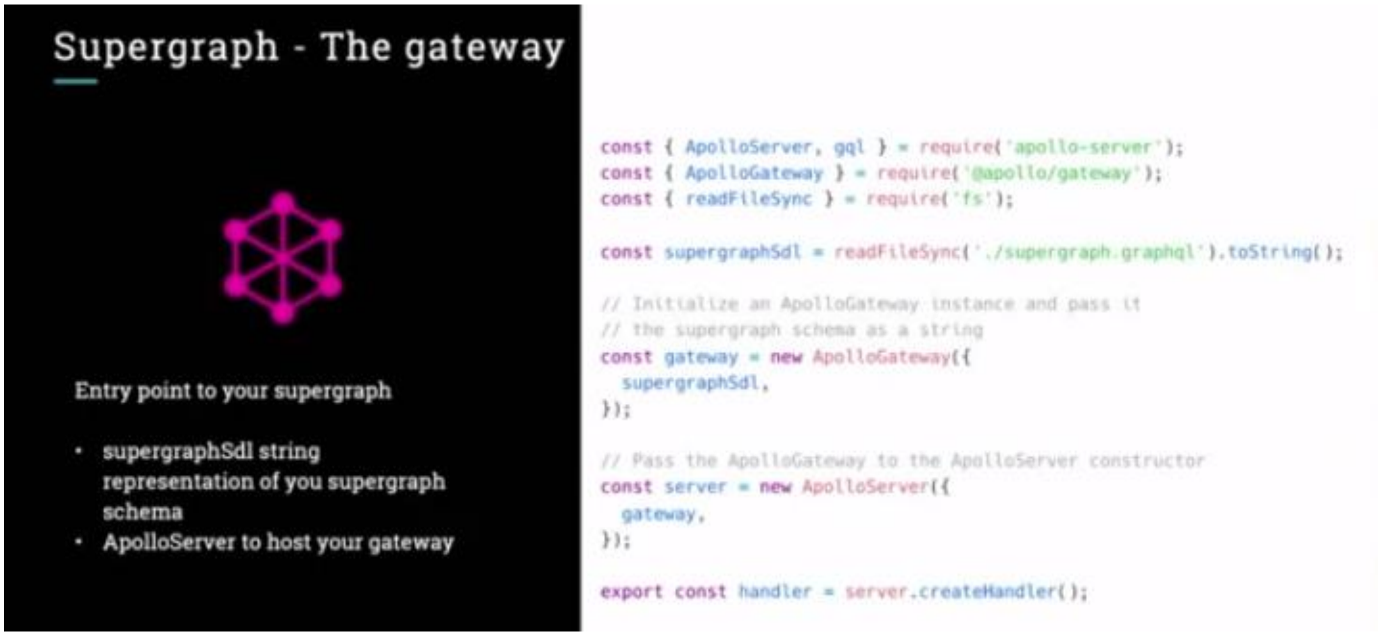


Supergraph - Query Execution & The Query Plan

```
query Movies {
  movie(id: "1") {
    id
    title
    reviews {
      id
      rating
      comments
    }
  }
}
```

```
type Movie @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String
}
```

```
type Movie @key(fields: "id") @extends {
  id: ID! @external
  reviews: [Review]
}

type Review @key(fields: "id") {
  id: ID!
  rating: Float
  comments: String
  movie: Movie
}
```

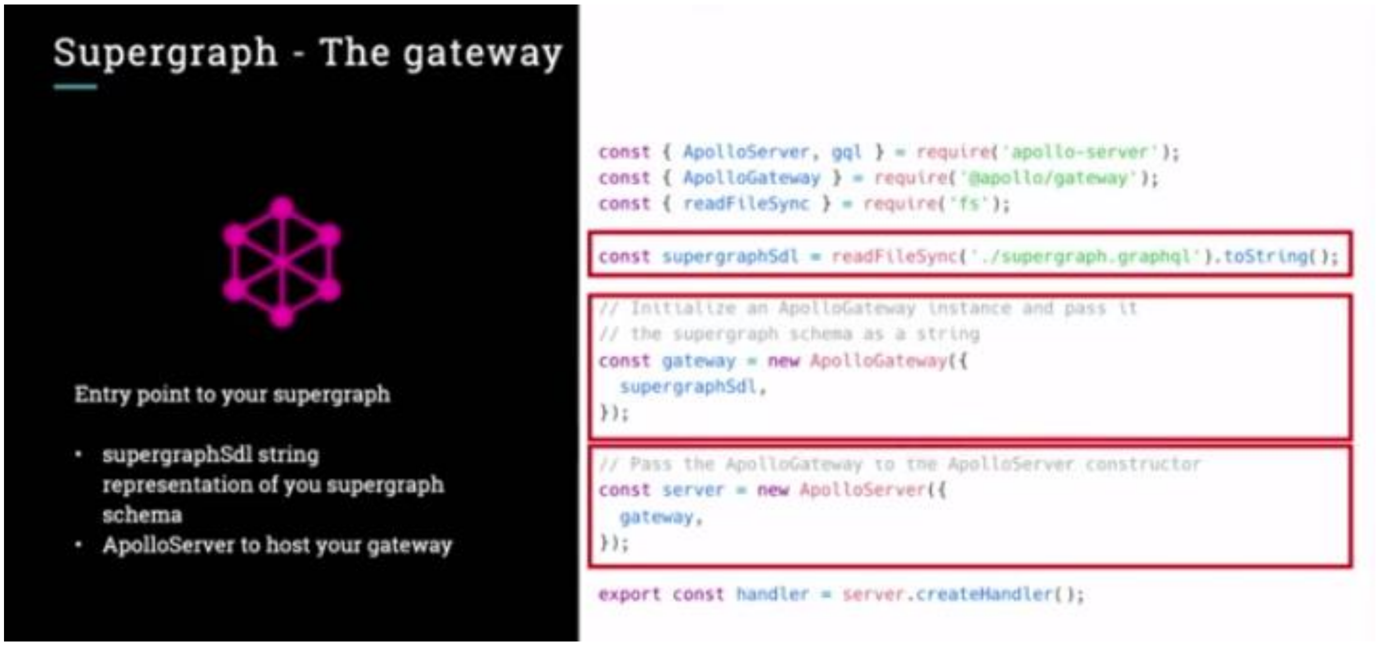


Supergraph - The gateway

Entry point to your supergraph

- supergraphSdl string representation of you supergraph schema
- ApolloServer to host your gateway

```
const { ApolloServer, gql } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { readFileSync } = require('fs');

const supergraphSdl = readFileSync('./supergraph.graphql').toString();

// Initialize an ApolloGateway instance and pass it
// the supergraph schema as a string
const gateway = new ApolloGateway({
  supergraphSdl,
});

// Pass the ApolloGateway to the ApolloServer constructor
const server = new ApolloServer({
  gateway,
});

export const handler = server.createHandler();
```

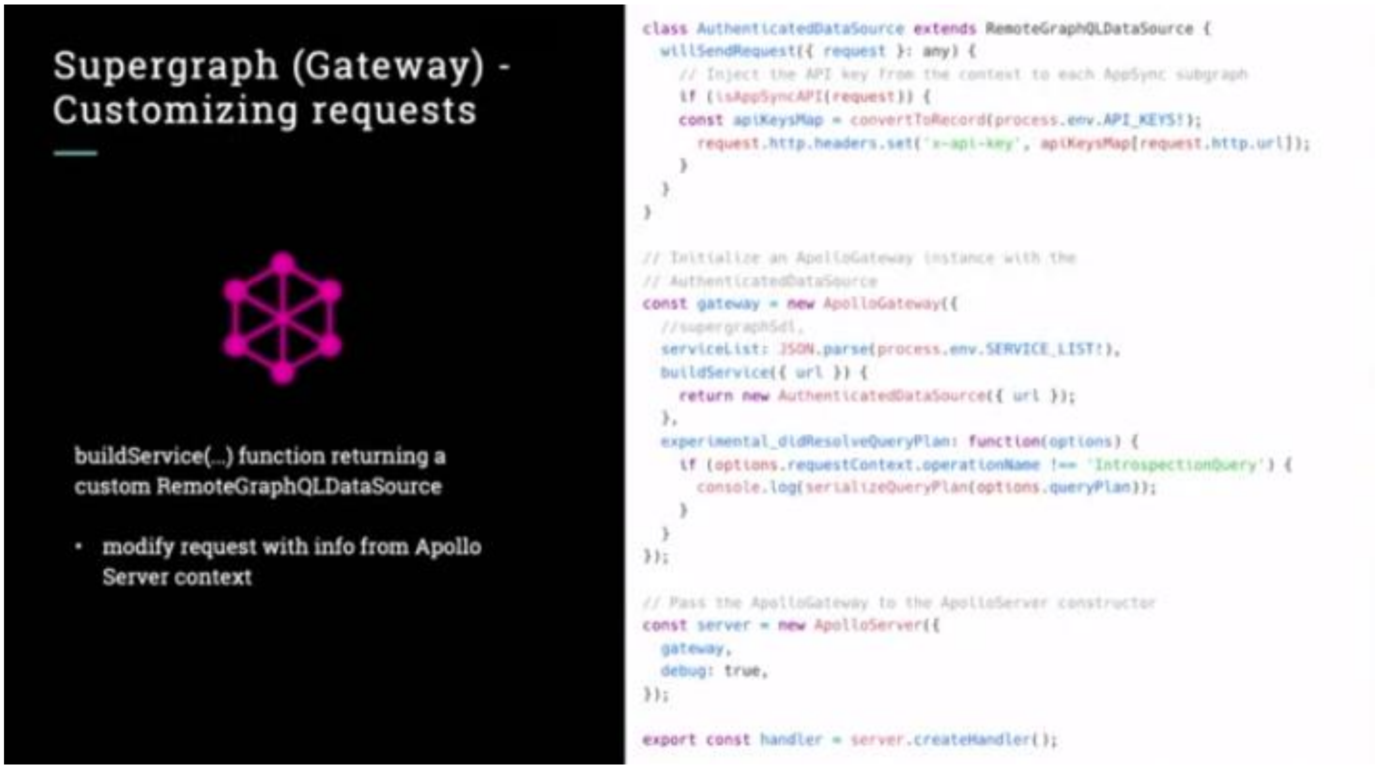


Supergraph - The gateway

Entry point to your supergraph

- supergraphSdl string representation of you supergraph schema
- ApolloServer to host your gateway

```
const { ApolloServer, gql } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { readFileSync } = require('fs');

const supergraphSdl = readFileSync('./supergraph.graphql').toString();

// Initialize an ApolloGateway instance and pass it
// the supergraph schema as a string
const gateway = new ApolloGateway({
  supergraphSdl,
});

// Pass the ApolloGateway to the ApolloServer constructor
const server = new ApolloServer({
  gateway,
});

export const handler = server.createHandler();
```

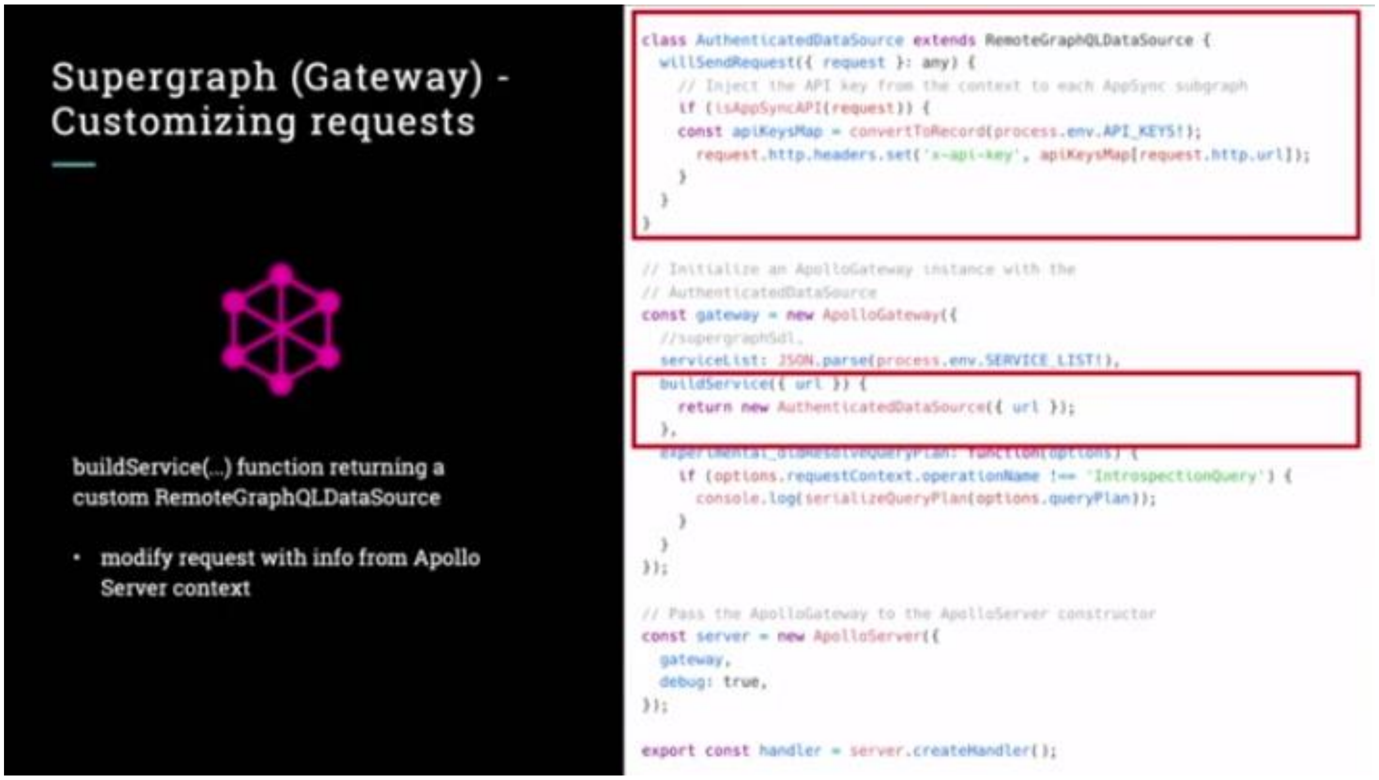


Supergraph (Gateway) - Customizing requests

buildService(...) function returning a custom RemoteGraphQLDataSource

- modify request with info from Apollo Server context

```
class AuthenticatedDataSource extends RemoteGraphQLDataSource {
  willSendRequest({ request }: any) {
    // Inject the API key from the context to each AppSync subgraph
    if (isAppSyncAPI(request)) {
    const apiKeysMap = convertToRecord(process.env.API_KEYS!);
      request.http.headers.set('x-api-key', apiKeysMap[request.http.url]);
    }
  }
}

// Initialize an ApolloGateway instance with the
// AuthenticatedDataSource
const gateway = new ApolloGateway({
  //supergraphSdl,
  serviceList: JSON.parse(process.env.SERVICE_LIST!),
  buildService({ url }) {
    return new AuthenticatedDataSource({ url });
  },
  experimental_didResolveQueryPlan: function(options) {
    if (options.requestContext.operationName !== 'IntrospectionQuery') {
      console.log(serializeQueryPlan(options.queryPlan));
    }
  }
});

// Pass the ApolloGateway to the ApolloServer constructor
const server = new ApolloServer({
  gateway,
  debug: true,
});

export const handler = server.createHandler();
```

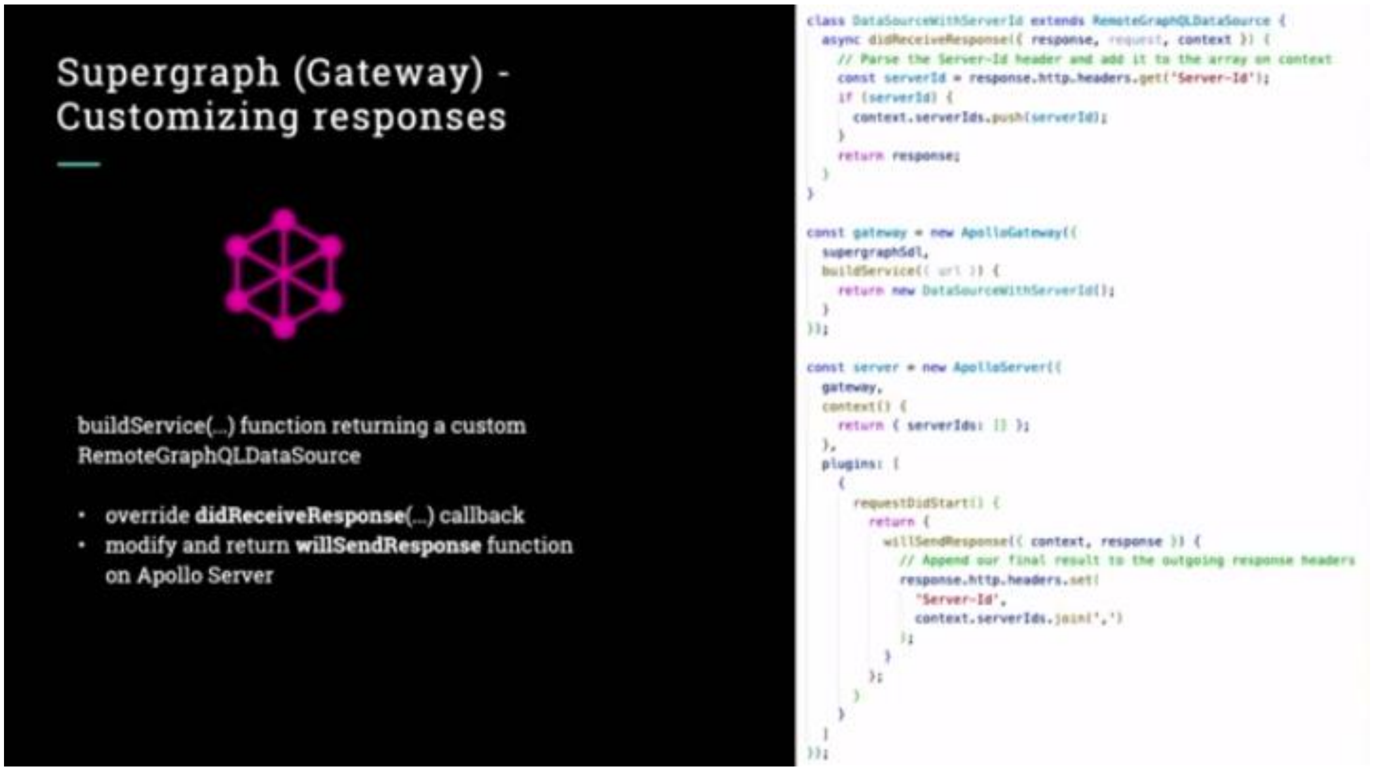


Supergraph (Gateway) - Customizing requests

buildService(...) function returning a custom RemoteGraphQLDataSource

- modify request with info from Apollo Server context

```
class AuthenticatedDataSource extends RemoteGraphQLDataSource {
  willSendRequest({ request }: any) {
    // Inject the API key from the context to each AppSync subgraph
    if (isAppSyncAPI(request)) {
    const apiKeysMap = convertToRecord(process.env.API_KEYS!);
      request.http.headers.set('x-api-key', apiKeysMap[request.http.url]);
    }
  }
}

// Initialize an ApolloGateway instance with the
// AuthenticatedDataSource
const gateway = new ApolloGateway({
  //supergraphSdl,
  serviceList: JSON.parse(process.env.SERVICE_LIST!),
  buildService({ url }) {
    return new AuthenticatedDataSource({ url });
  },
  experimental_didResolveQueryPlan: function(options) {
    if (options.requestContext.operationName !== 'IntrospectionQuery') {
      console.log(serializeQueryPlan(options.queryPlan));
    }
  }
});

// Pass the ApolloGateway to the ApolloServer constructor
const server = new ApolloServer({
  gateway,
  debug: true,
});

export const handler = server.createHandler();
```

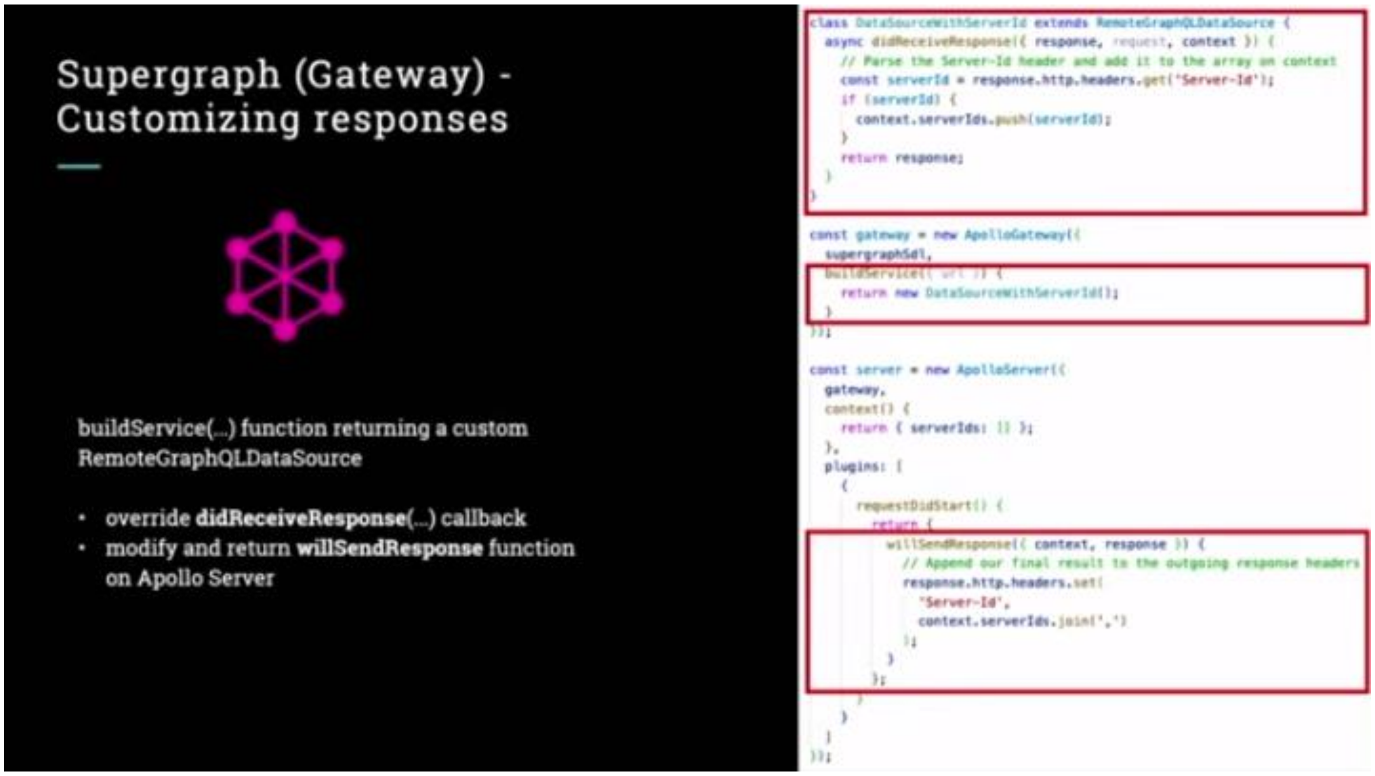


Supergraph (Gateway) - Customizing responses

buildService(...) function returning a custom RemoteGraphQLDataSource

- override **didReceiveResponse**(...) callback
- modify and return **willSendResponse** function on Apollo Server

```
class DataSourceWithServerId extends RemoteGraphQLDataSource {
  async didReceiveResponse({ response, request, context }) {
    // Parse the Server-Id header and add it to the array on context
    const serverId = response.http.headers.get('Server-Id');
    if (serverId) {
      context.serverIds.push(serverId);
    }
    return response;
  }
}

const gateway = new ApolloGateway({
  supergraphSdl,
  buildService({ url }) {
    return new DataSourceWithServerId();
  }
});

const server = new ApolloServer({
  gateway,
  context() {
    return { serverIds: [] };
  },
  plugins: [
    {
      requestDidStart() {
        return {
          willSendResponse({ context, response }) {
            // Append our final result to the outgoing response headers
            response.http.headers.set(
              'Server-Id',
              context.serverIds.join(',')
            );
          }
        };
      }
    }
  ]
});
```

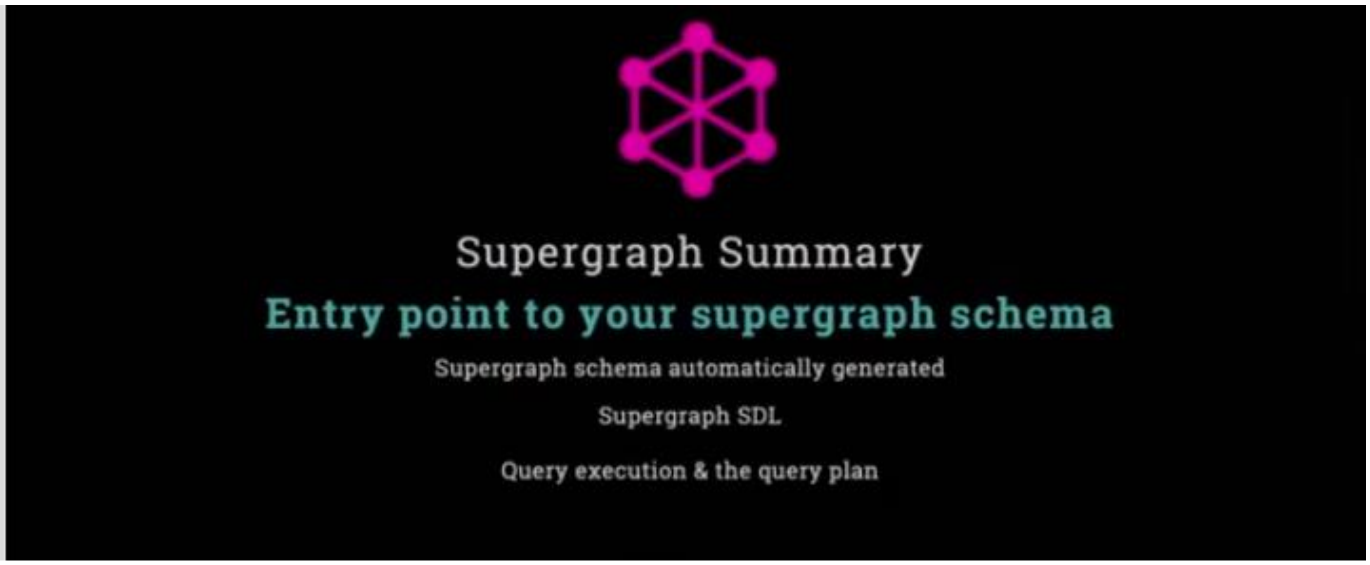


Supergraph (Gateway) - Customizing responses

buildService(...) function returning a custom RemoteGraphQLDataSource

- override **didReceiveResponse**(...) callback
- modify and return **willSendResponse** function on Apollo Server

```
class DataSourceWithServerId extends RemoteGraphQLDataSource {
  async didReceiveResponse({ response, request, context }) {
    // Parse the Server-Id header and add it to the array on context
    const serverId = response.http.headers.get('Server-Id');
    if (serverId) {
      context.serverIds.push(serverId);
    }
    return response;
  }
}

const gateway = new ApolloGateway({
  supergraphSdl,
  buildService({ url }) {
    return new DataSourceWithServerId();
  }
});

const server = new ApolloServer({
  gateway,
  context() {
    return { serverIds: [] };
  },
  plugins: [
    {
      requestDidStart() {
        return {
          willSendResponse({ context, response }) {
            // Append our final result to the outgoing response headers
            response.http.headers.set(
              'Server-Id',
              context.serverIds.join(',')
            );
          }
        };
      }
    }
  ]
});
```



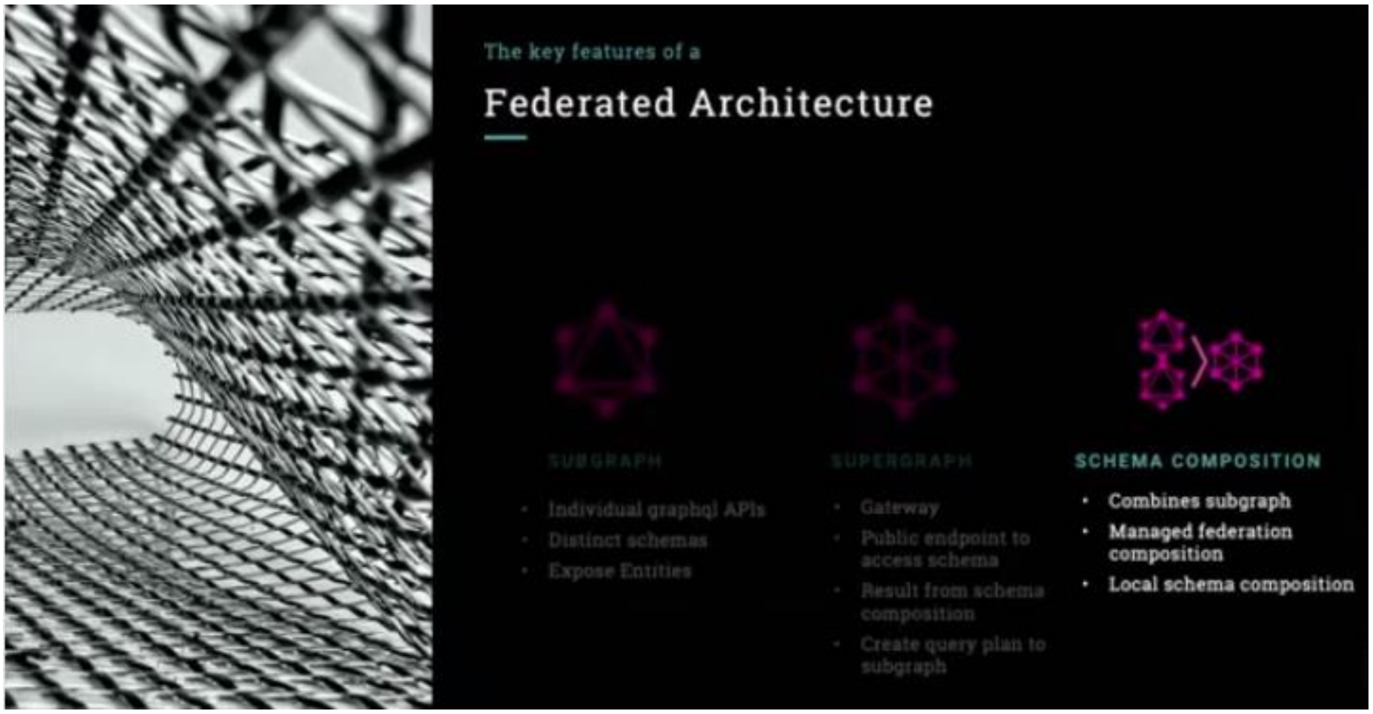
Supergraph Summary

Entry point to your supergraph schema

Supergraph schema automatically generated

Supergraph SDL

Query execution & the query plan



The key features of a

Federated Architecture

SUBGRAPH

- Individual graphql APIs
- Distinct schemas
- Expose Entities

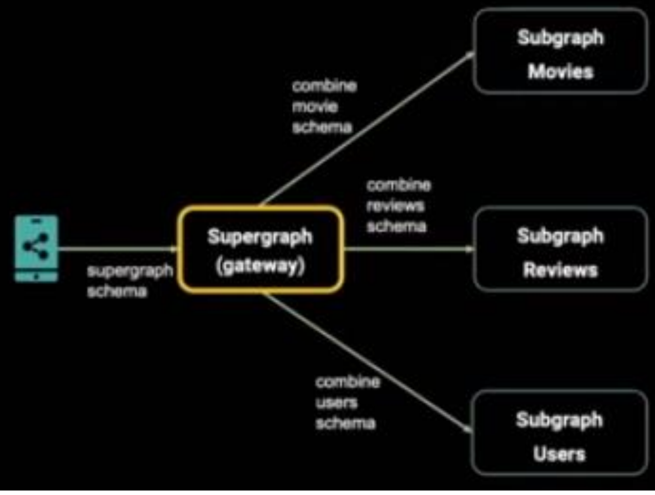
SUPERGRAPH

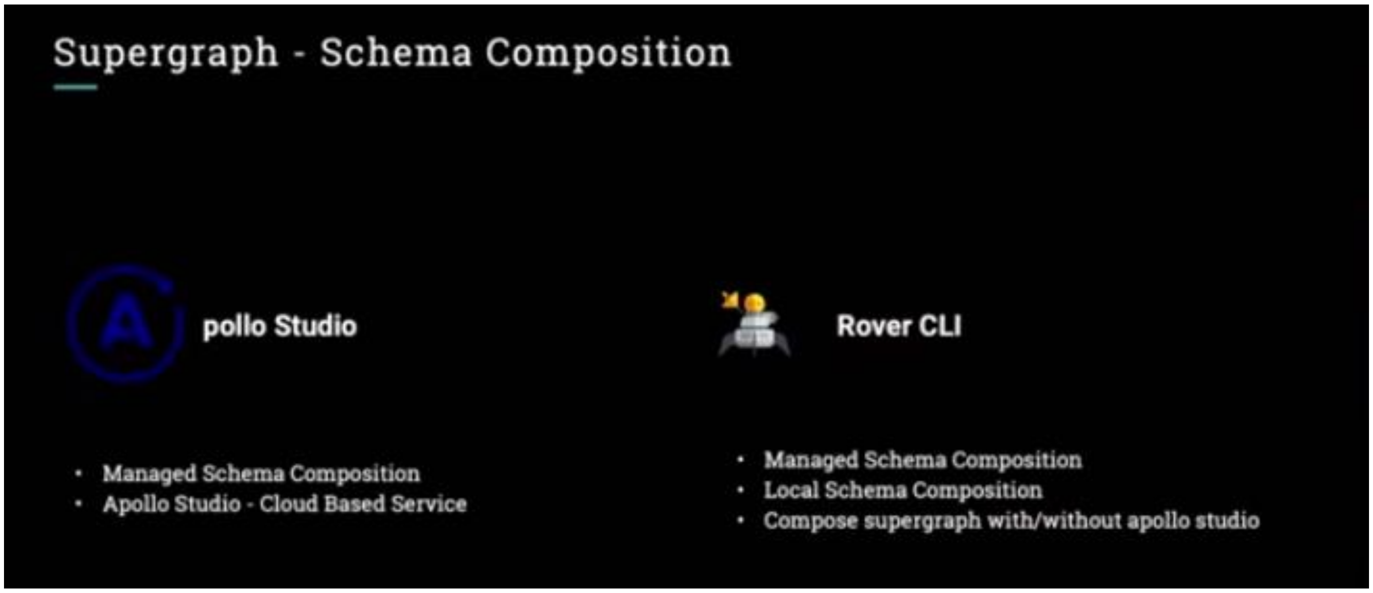
- Gateway
- Public endpoint to access schema
- Result from schema composition
- Create query plan to subgraph

SCHEMA COMPOSITION

- Combines subgraph
- Managed federation composition
- Local schema composition

Schema Composition





Supergraph - Schema Composition

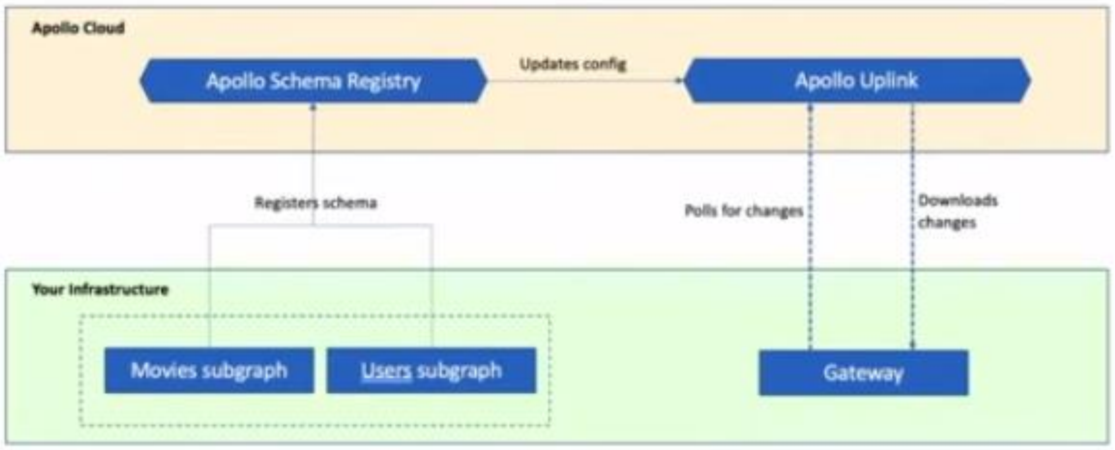
pollo Studio

- Managed Schema Composition
- Apollo Studio - Cloud Based Service

Rover CLI

- Managed Schema Composition
- Local Schema Composition
- Compose supergraph with/without apollo studio

Schema Composition - Managed Federation



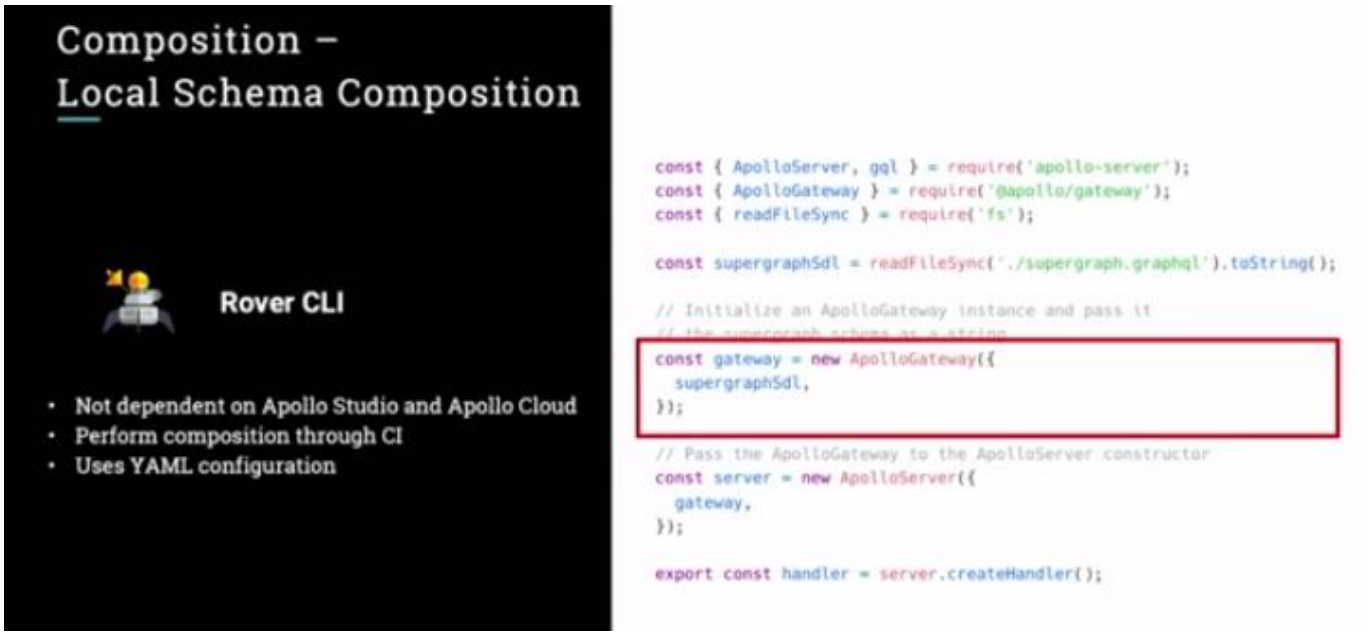
Schema Composition - Apollo Studio

Composition – Local Schema Composition

Rover CLI

- Not dependent on Apollo Studio and Apollo Cloud
- Perform composition through CI
- Uses YAML configuration

```
const { ApolloServer, gql } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { readFileSync } = require('fs');

const supergraphSdl = readFileSync('./supergraph.graphql').toString();

// Initialize an ApolloGateway instance and pass it
// the supergraph schema as a string
const gateway = new ApolloGateway({
  supergraphSdl,
});

// Pass the ApolloGateway to the ApolloServer constructor
const server = new ApolloServer({
  gateway,
});

export const handler = server.createHandler();
```

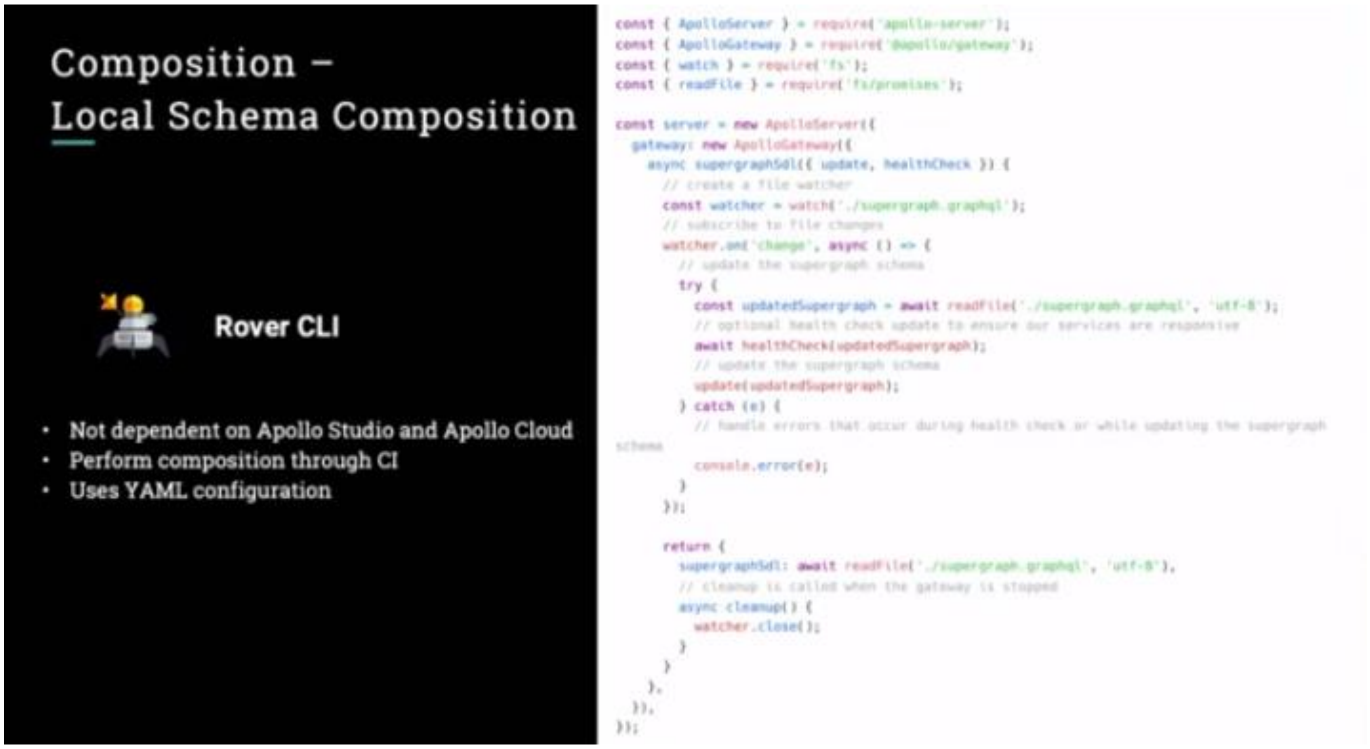


Composition – Local Schema Composition

Rover CLI

- Not dependent on Apollo Studio and Apollo Cloud
- Perform composition through CI
- Uses YAML configuration

```
const { ApolloServer, gql } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { readFileSync } = require('fs');

const supergraphSdl = readFileSync('./supergraph.graphql').toString();

// Initialize an ApolloGateway instance and pass it
// the supergraph schema as a string
const gateway = new ApolloGateway({
  supergraphSdl,
});

// Pass the ApolloGateway to the ApolloServer constructor
const server = new ApolloServer({
  gateway,
});

export const handler = server.createHandler();
```

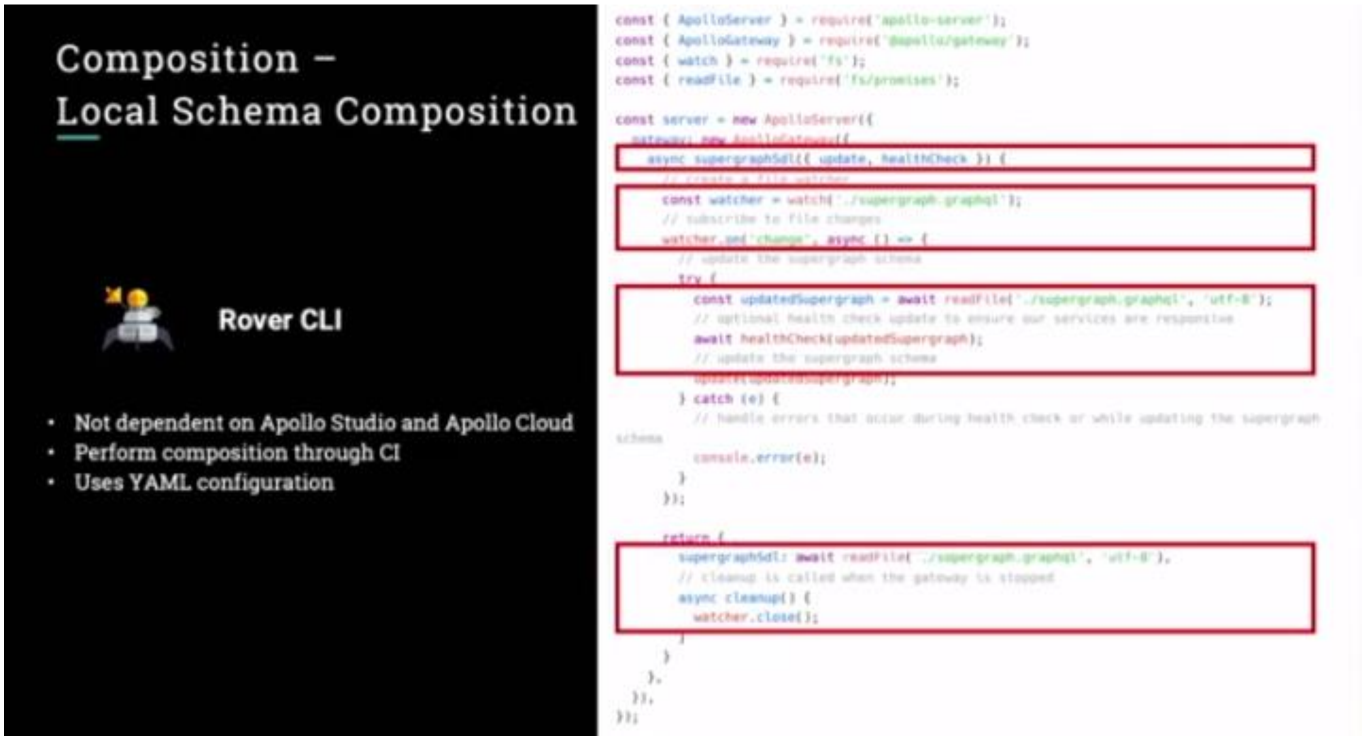


Composition – Local Schema Composition

Rover CLI

- Not dependent on Apollo Studio and Apollo Cloud
- Perform composition through CI
- Uses YAML configuration

```
const { ApolloServer } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { watch } = require('fs');
const { readFile } = require('fs/promises');

const server = new ApolloServer({
  gateway: new ApolloGateway({
    async supergraphSdl({ update, healthCheck }) {
      // create a file watcher
      const watcher = watch('./supergraph.graphql');
      // subscribe to file changes
      watcher.on('change', async () => {
        // update the supergraph schema
        try {
          const updatedSupergraph = await readFile('./supergraph.graphql', 'utf-8');
          // optional health check update to ensure our services are responsive
          await healthCheck(updatedSupergraph);
          // update the supergraph schema
          update(updatedSupergraph);
        } catch (e) {
          // handle errors that occur during health check or while updating the supergraph
schema
          console.error(e);
        }
      });

      return {
        supergraphSdl: await readFile('./supergraph.graphql', 'utf-8'),
        // cleanup is called when the gateway is stopped
        async cleanup() {
          watcher.close();
        }
      }
    },
  }),
});
```

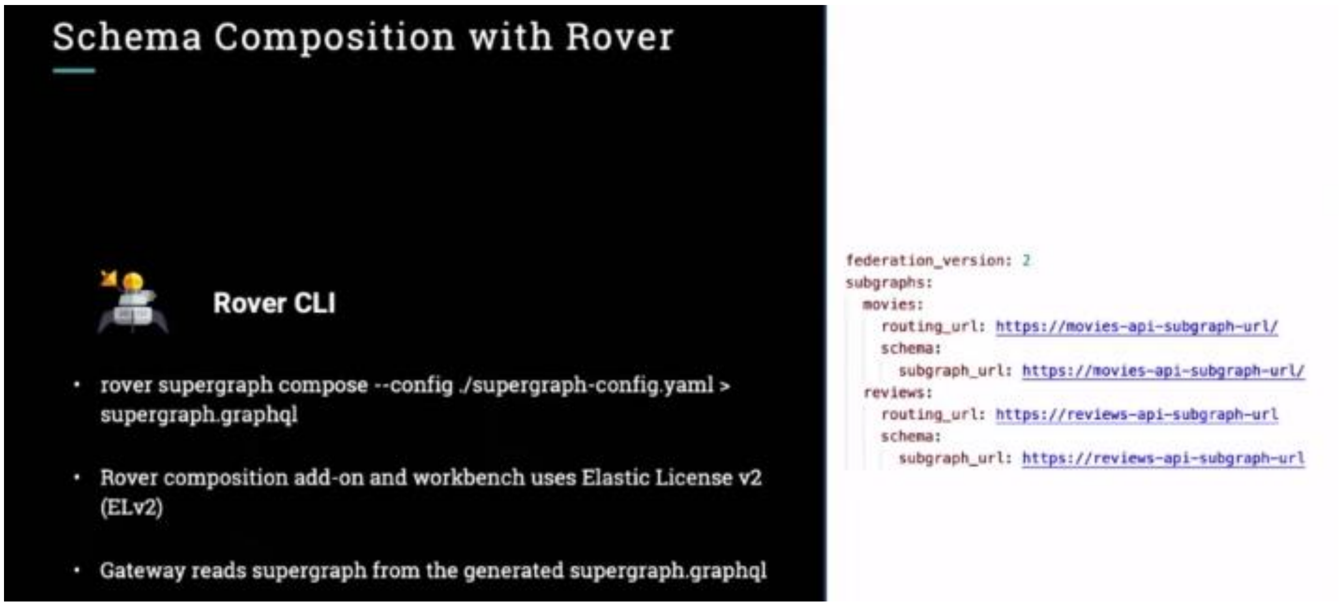


Composition – Local Schema Composition

Rover CLI

- Not dependent on Apollo Studio and Apollo Cloud
- Perform composition through CI
- Uses YAML configuration

```
const { ApolloServer } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { watch } = require('fs');
const { readFile } = require('fs/promises');

const server = new ApolloServer({
  gateway: new ApolloGateway({
    async supergraphSdl({ update, healthCheck }) {
      // create a file watcher
      const watcher = watch('./supergraph.graphql');
      // subscribe to file changes
      watcher.on('change', async () => {
        // update the supergraph schema
        try {
          const updatedSupergraph = await readFile('./supergraph.graphql', 'utf-8');
          // optional health check update to ensure our services are responsive
          await healthCheck(updatedSupergraph);
          // update the supergraph schema
          update(updatedSupergraph);
        } catch (e) {
          // handle errors that occur during health check or while updating the supergraph
schema
          console.error(e);
        }
      });

      return {
        supergraphSdl: await readFile('./supergraph.graphql', 'utf-8'),
        // cleanup is called when the gateway is stopped
        async cleanup() {
          watcher.close();
        }
      }
    },
  }),
});
```

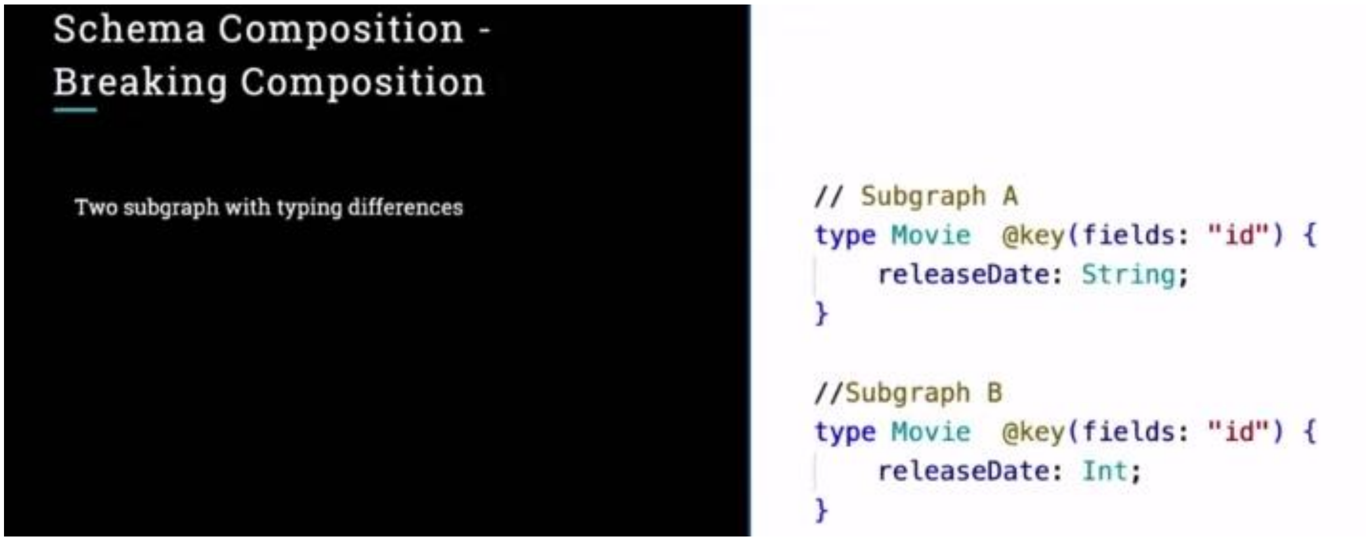


Schema Composition with Rover

Rover CLI

- rover supergraph compose --config ./supergraph-config.yaml > supergraph.graphql
- Rover composition add-on and workbench uses Elastic License v2 (ELv2)
- Gateway reads supergraph from the generated supergraph.graphql

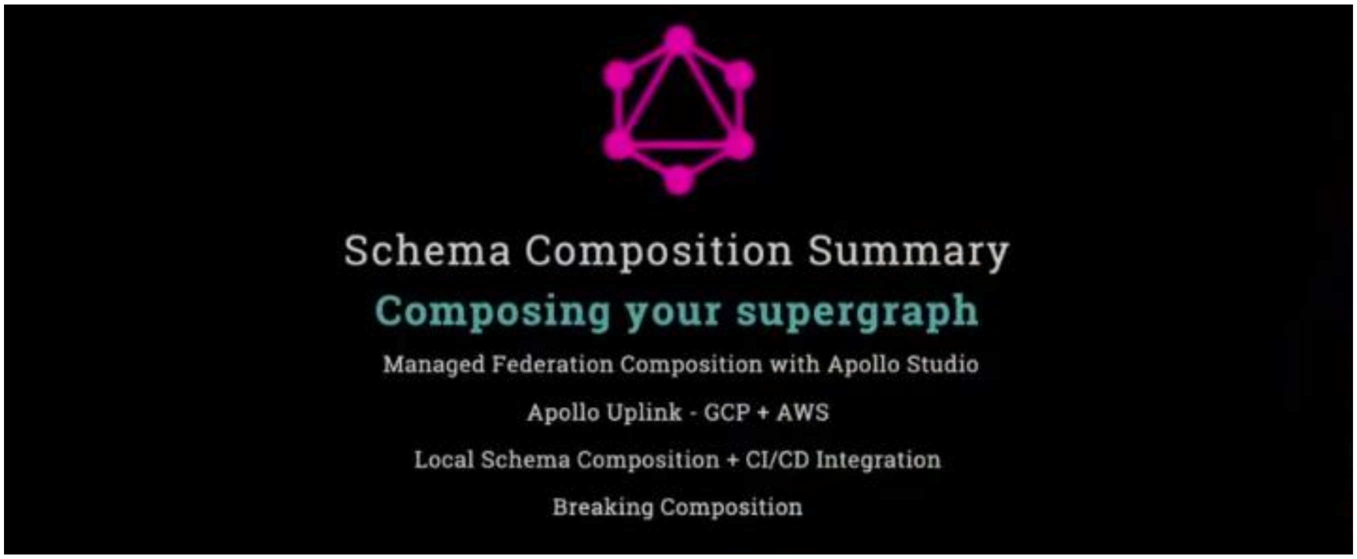
```yaml
federation_version: 2
subgraphs:
  movies:
    routing_url: https://movies-api-subgraph-url/
    schema:
      subgraph_url: https://movies-api-subgraph-url/
  reviews:
    routing_url: https://reviews-api-subgraph-url
    schema:
      subgraph_url: https://reviews-api-subgraph-url
```



Schema Composition - Breaking Composition

Two subgraph with typing differences

```graphql
// Subgraph A
type Movie  @key(fields: "id") {
    releaseDate: String;
}

//Subgraph B
type Movie  @key(fields: "id") {
    releaseDate: Int;
}
```



Schema Composition Summary

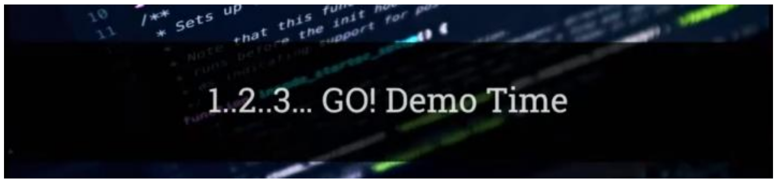
Composing your supergraph

Managed Federation Composition with Apollo Studio

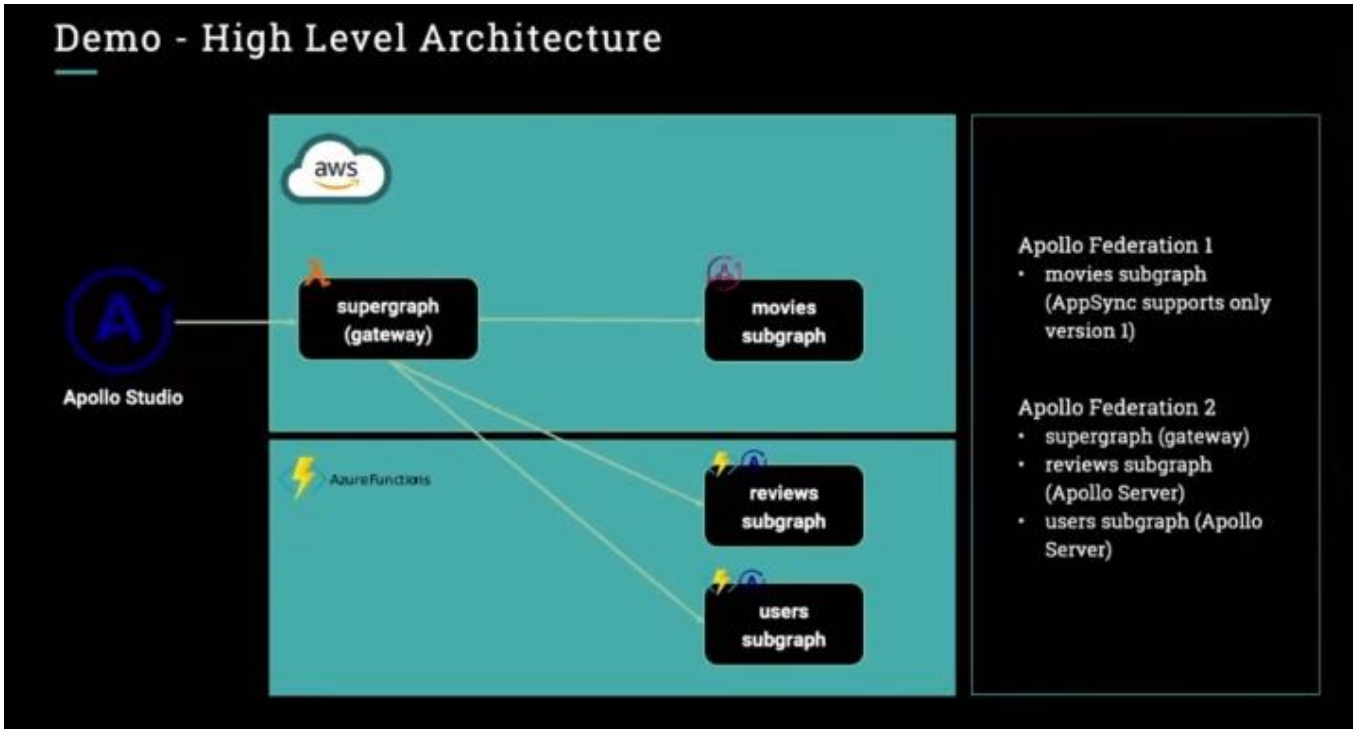
Apollo Uplink - GCP + AWS

Local Schema Composition + CI/CD Integration

Breaking Composition



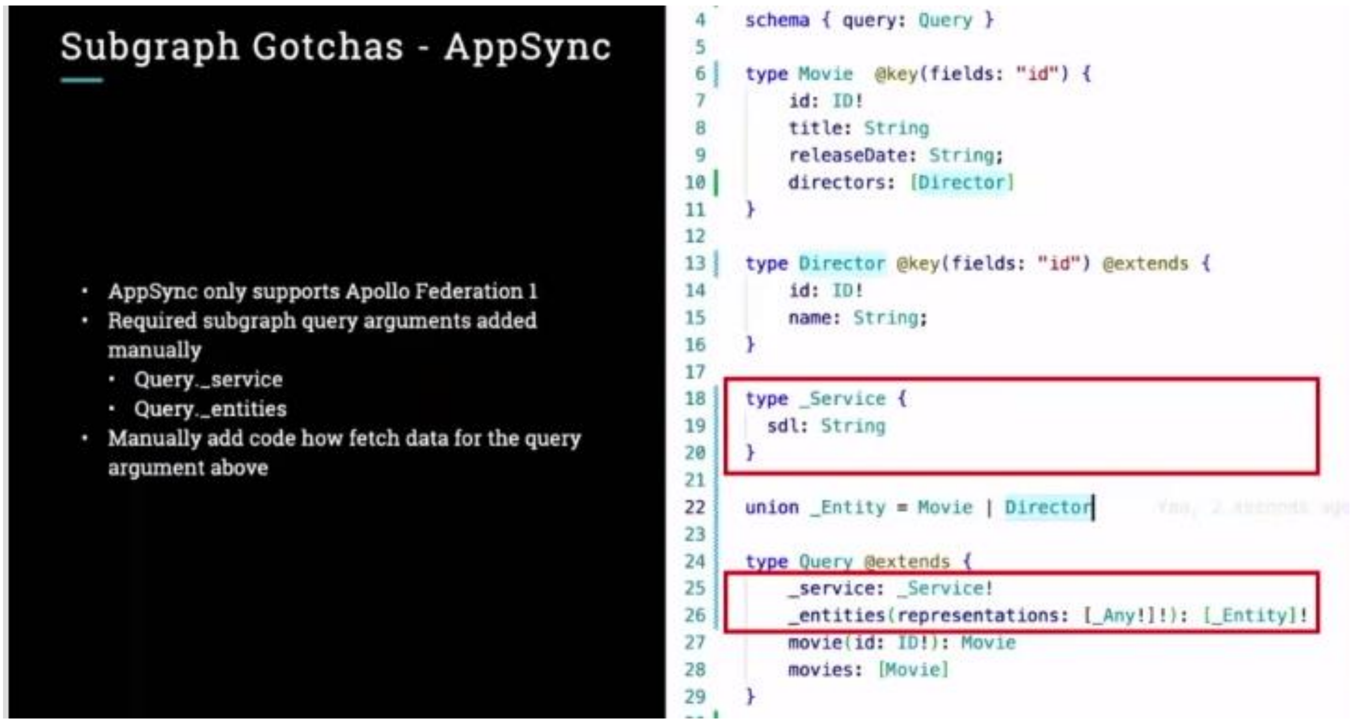
1..2..3... GO! Demo Time



Demo - High Level Architecture

Apollo Federation 1
- movies subgraph (AppSync supports only version 1)

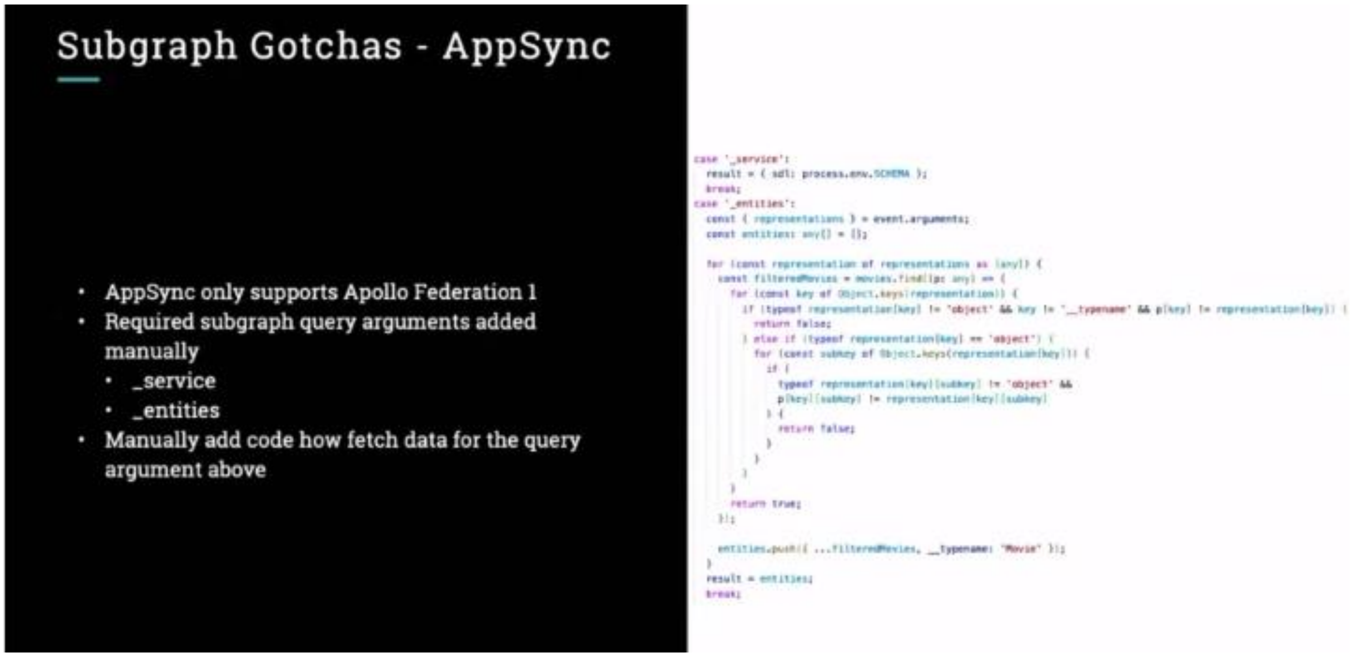
Apollo Federation 2
- supergraph (gateway)
- reviews subgraph (Apollo Server)
- users subgraph (Apollo Server)



Subgraph Gotchas - AppSync

- AppSync only supports Apollo Federation 1
- Required subgraph query arguments added manually
  - Query._service
  - Query._entities
- Manually add code how fetch data for the query argument above

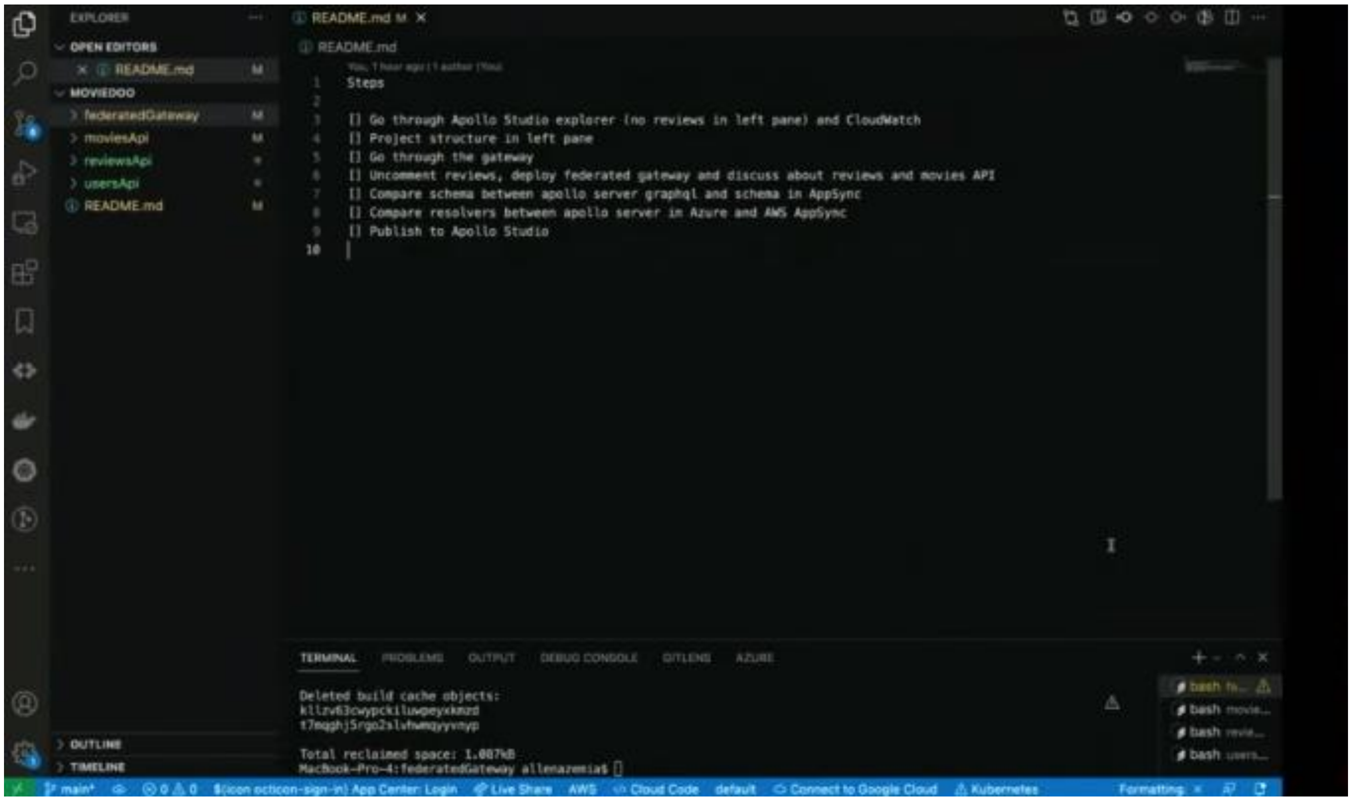
```graphql
schema { query: Query }

type Movie  @key(fields: "id") {
    id: ID!
    title: String
    releaseDate: String;
    directors: [Director]
}

type Director @key(fields: "id") @extends {
    id: ID!
    name: String;
}

type _Service {
  sdl: String
}

union _Entity = Movie | Director

type Query @extends {
    _service: _Service!
    _entities(representations: [_Any!]!): [_Entity]!
    movie(id: ID!): Movie
    movies: [Movie]
}
```



Subgraph Gotchas - AppSync

- AppSync only supports Apollo Federation 1
- Required subgraph query arguments added manually
  - _service
  - _entities
- Manually add code how fetch data for the query argument above

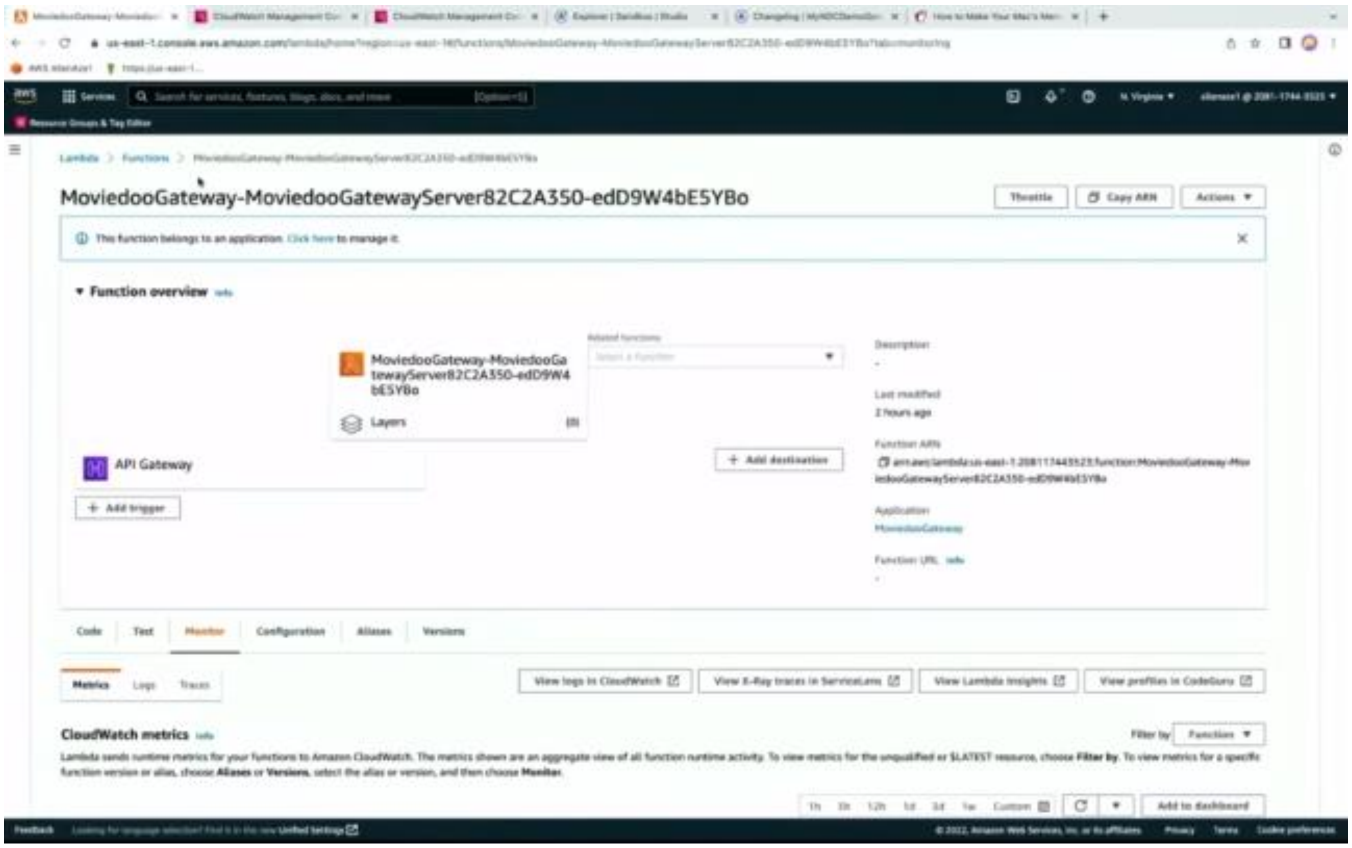
```
case '_service':
  result = { sdl: process.env.SCHEMA };
  break;
case '_entities':
  const { representations } = event.arguments;
  const entities: any[] = [];

  for (const representation of representations as [any]) {
    const filteredMovies = movies.find((p: any) => {
      for (const key of Object.keys(representation)) {
        if (typeof representation[key] != 'object' && key != '__typename' && p[key] != representation[key]) {
          return false;
        } else if (typeof representation[key] == 'object') {
          for (const subkey of Object.keys(representation[key])) {
            if (
              typeof representation[key][subkey] != 'object' &&
              p[key][subkey] != representation[key][subkey]
            ) {
              return false;
            }
          }
        }
      }
      return true;
    });

    entities.push({ ...filteredMovies, __typename: "Movie" });
  }
  result = entities;
  break;
```



README.md

```
Steps

[] Go through Apollo Studio explorer (no reviews in left pane) and CloudWatch
[] Project structure in left pane
[] Go through the gateway
[] Uncomment reviews, deploy federated gateway and discuss about reviews and movies API
[] Compare schema between apollo server graphql and schema in AppSync
[] Compare resolvers between apollo server in Azure and AWS AppSync
[] Publish to Apollo Studio
```

```
Deleted build cache objects:
kllzv63cwypckiluwpeyxkmzd
t7mqghj5rgo2slvhwmqyyvnyp

Total reclaimed space: 1.007kB
MacBook-Pro-4:federatedGateway allenazemia$ 
```



MoviedooGateway-MoviedooGatewayServer82C2A350-edD9W4bE5YBo

This function belongs to an application. Click here to manage it.

Function overview

MoviedooGateway-MoviedooGatewayServer82C2A350-edD9W4bE5YBo

Description
-

Last modified
2 hours ago

Function ARN
arn:aws:lambda:us-east-1:208117445523:function:MoviedooGateway-MoviedooGatewayServer82C2A350-edD9W4bE5YBo

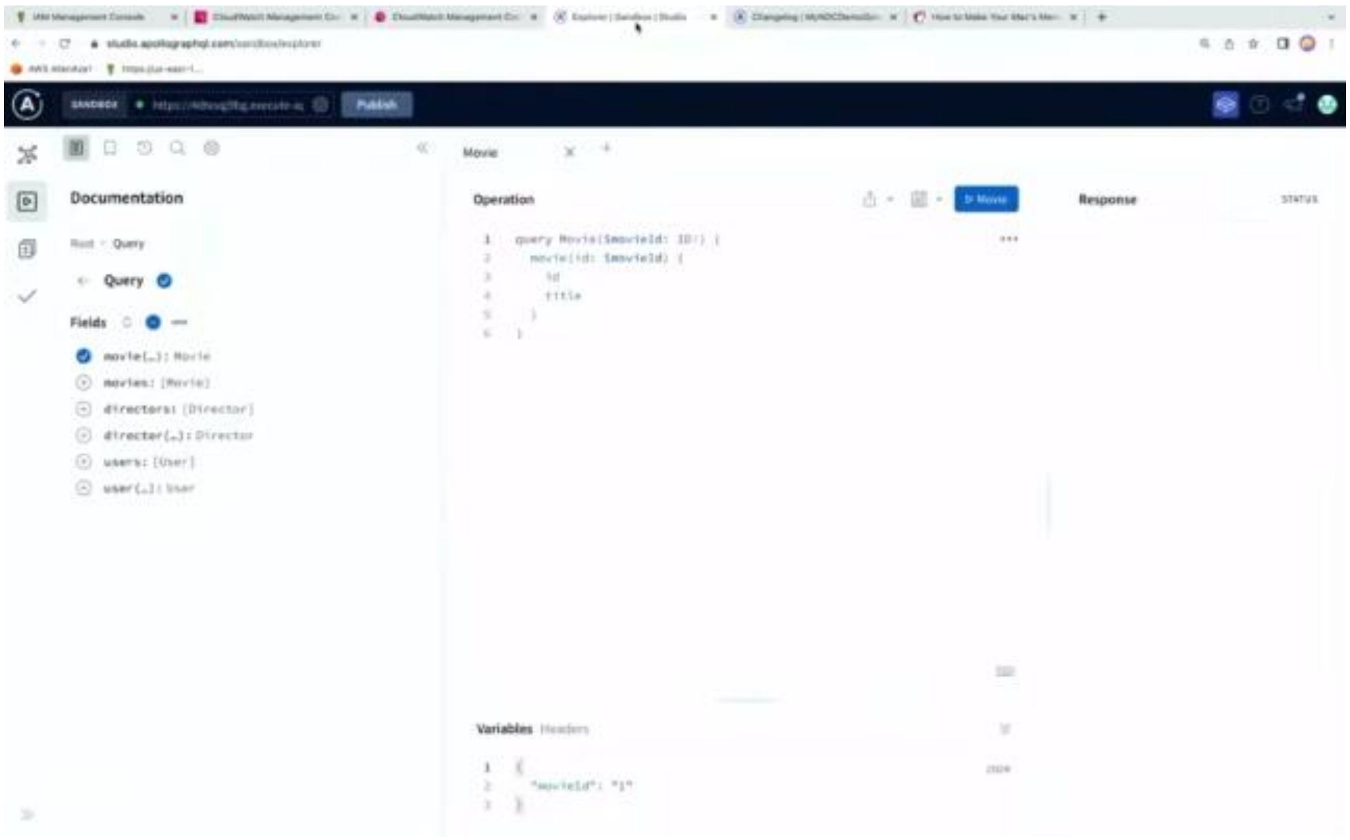
Application
MoviedooGateway

Function URL
-

CloudWatch metrics

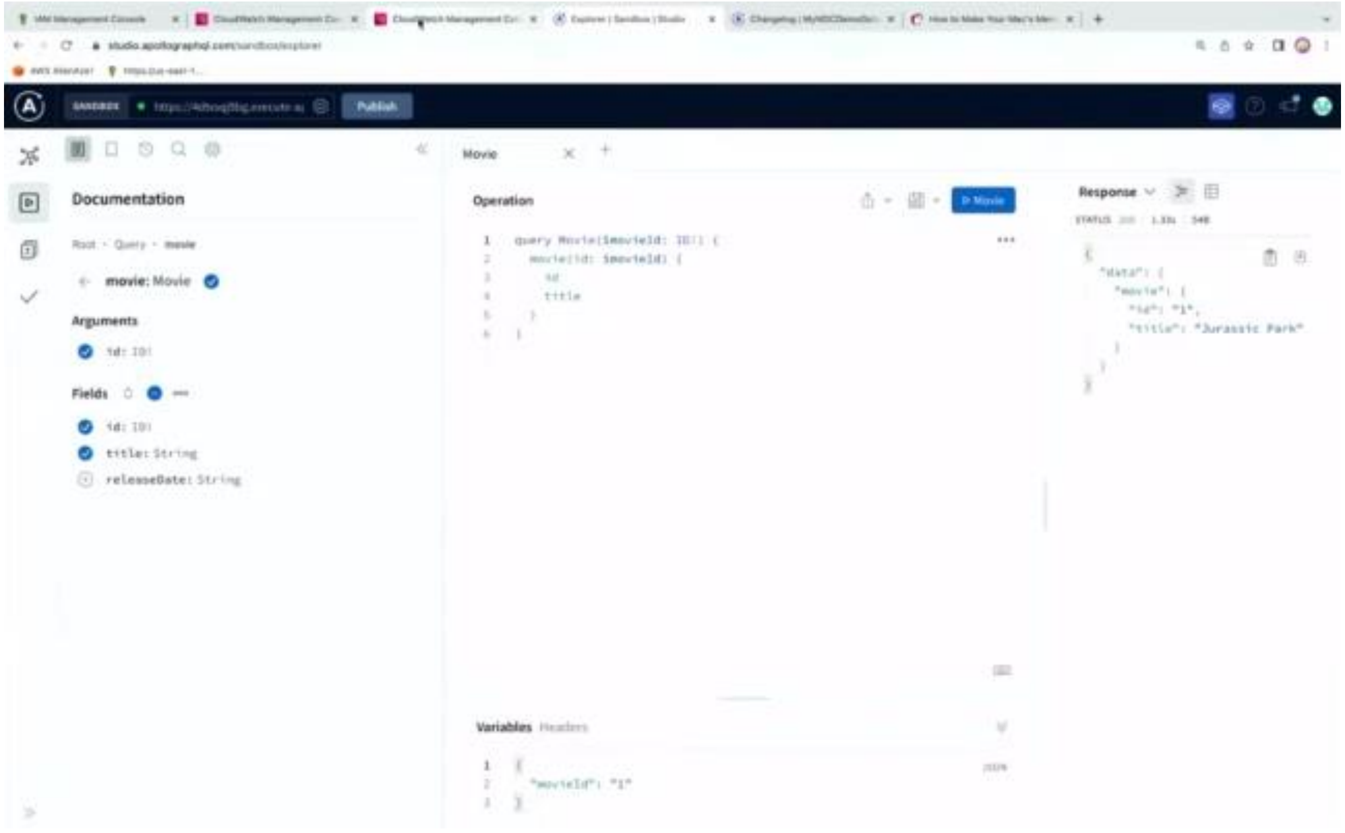
Lambda sends runtime metrics for your functions to Amazon CloudWatch. The metrics shown are an aggregate view of all function runtime activity. To view metrics for the unqualified or $LATEST resource, choose **Filter by**. To view metrics for a specific function version or alias, choose **Aliases** or **Versions**, select the alias or version, and then choose **Monitor**.

© 2022, Amazon Web Services, Inc. or its affiliates.



```
query Movie($movieId: ID!) {
  movie(id: $movieId) {
    id
    title
  }
}
```

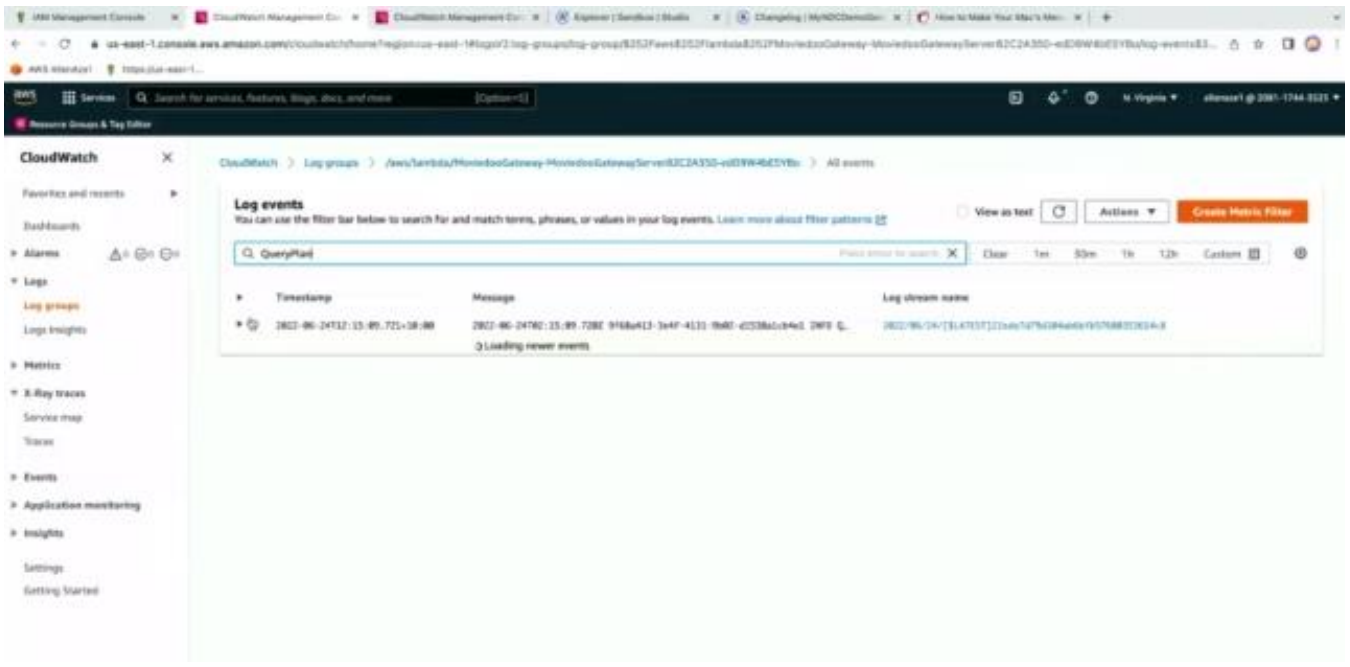
```
{
  "movieId": "1"
}
```



```
query Movie($movieId: ID!) {
  movie(id: $movieId) {
    id
    title
  }
}
```

```
{
  "data": {
    "movie": {
      "id": "1",
      "title": "Jurassic Park"
    }
  }
}
```

```
{
  "movieId": "1"
}
```



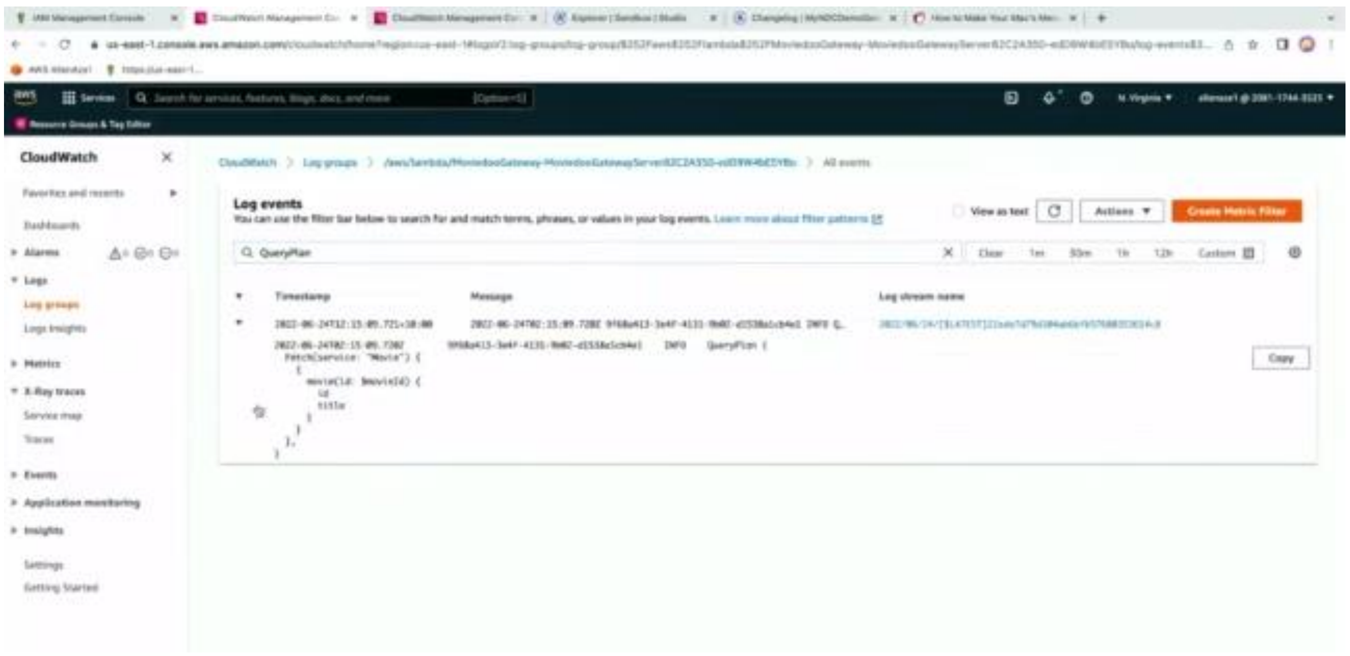
CloudWatch > Log groups > /aws/lambda/MoviedooGateway-MoviedooGatewayServer82C2A350-edD9W4bESYBo > All events

Log events

You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

QueryPlan

```
Timestamp                        Message                                                          Log stream name
2022-06-24T12:15:09.721+10:00    2022-06-24T02:15:09.728Z 9f68a413-3e4f-4131-9b02-d1538a1cb4e1 INFO Q...    2022/06/24/[$LATEST][illegible]
Loading newer events
```



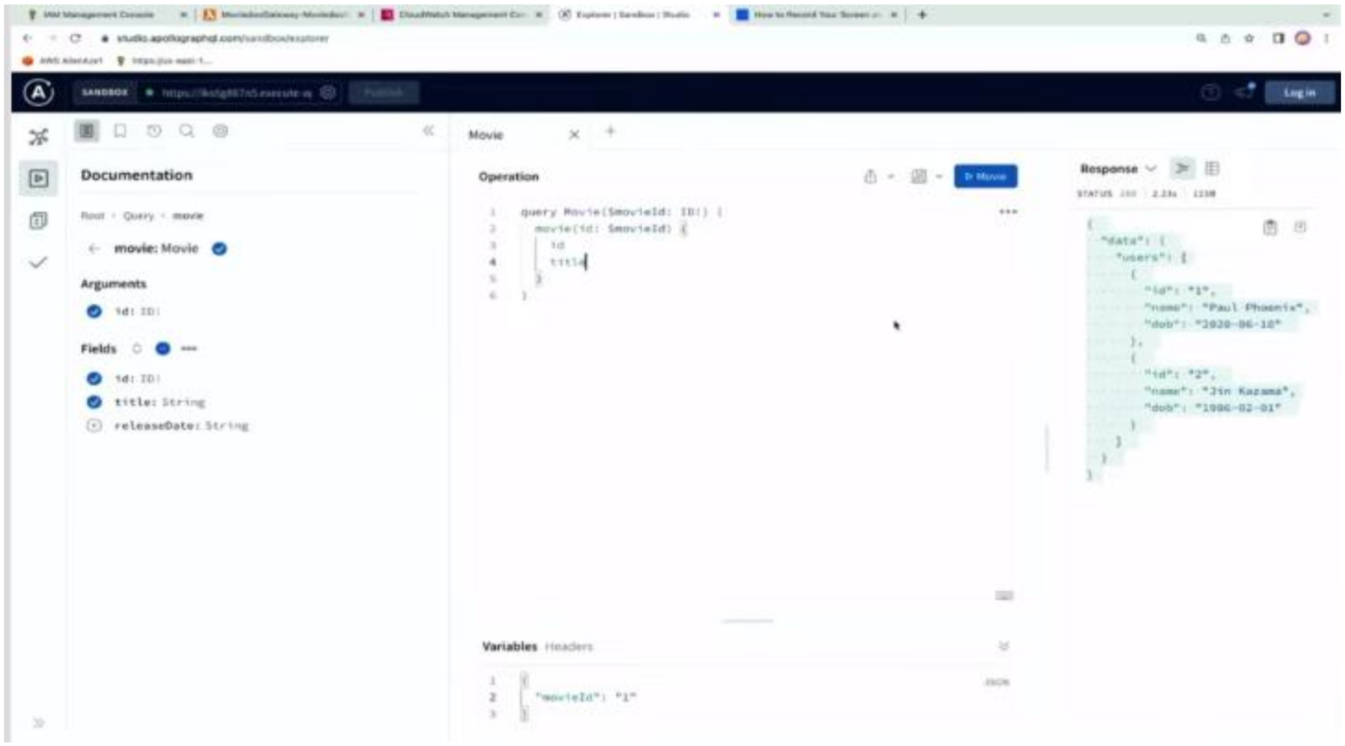
CloudWatch > Log groups > /aws/lambda/MoviedooGateway-MoviedooGatewayServer82C2A350-edD9W4bESYBo > All events

Log events

You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

QueryPlan

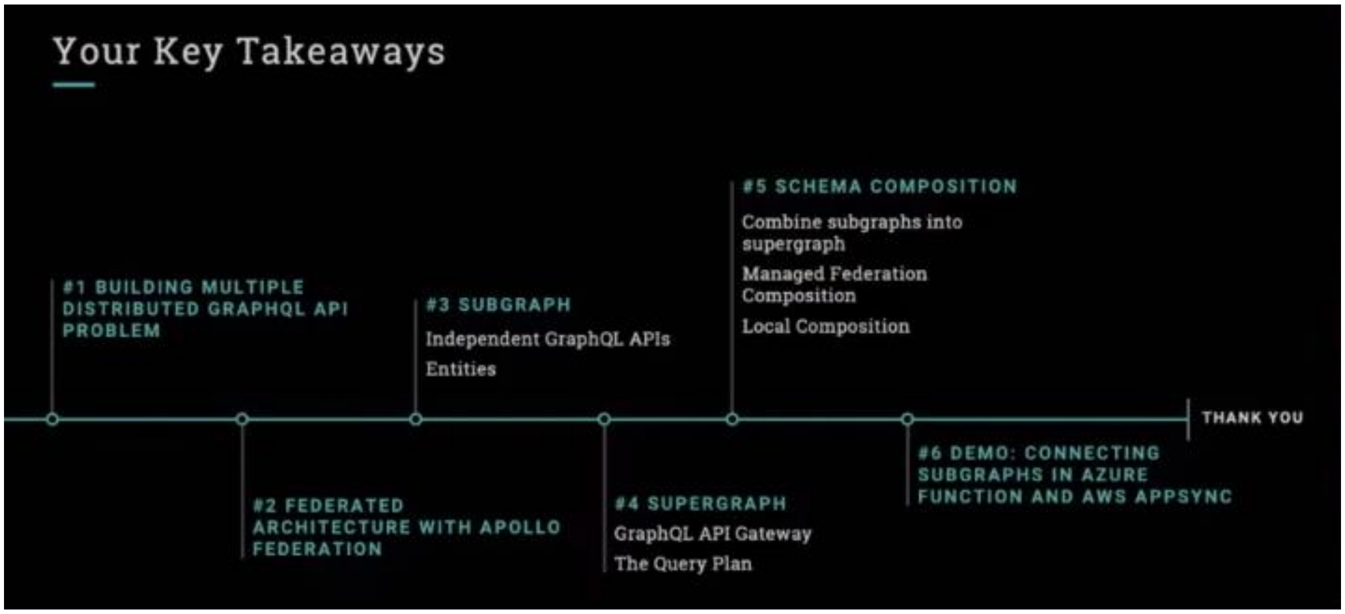
```
Timestamp                        Message                                                          Log stream name
2022-06-24T12:15:09.721+10:00    2022-06-24T02:15:09.728Z 9f68a413-3e4f-4131-9b02-d1538a1cb4e1 INFO Q...    2022/06/24/[$LATEST][illegible]
2022-06-24T02:15:09.728Z    9f68a413-3e4f-4131-9b02-d1538a1cb4e1    INFO    QueryPlan {
  Fetch(service: "Movie") {
    {
      movie(id: $movieId) {
        id
        title
      }
    }
  },
}
```



```
query Movie($movieId: ID!) {
  movie(id: $movieId) {
    id
    title
  }
}
```

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "name": "Paul Phoenix",
        "dob": "2020-06-10"
      },
      {
        "id": "2",
        "name": "Jin Kazama",
        "dob": "1996-02-01"
      }
    ]
  }
}
```

```
{
  "movieId": "1"
}
```



Your Key Takeaways

#1 BUILDING MULTIPLE DISTRIBUTED GRAPHQL API PROBLEM

#2 FEDERATED ARCHITECTURE WITH APOLLO FEDERATION

#3 SUBGRAPH

Independent GraphQL APIs

Entities

#4 SUPERGRAPH

GraphQL API Gateway

The Query Plan

#5 SCHEMA COMPOSITION

Combine subgraphs into supergraph

Managed Federation Composition

Local Composition

#6 DEMO: CONNECTING SUBGRAPHS IN AZURE FUNCTION AND AWS APPSYNC

THANK YOU