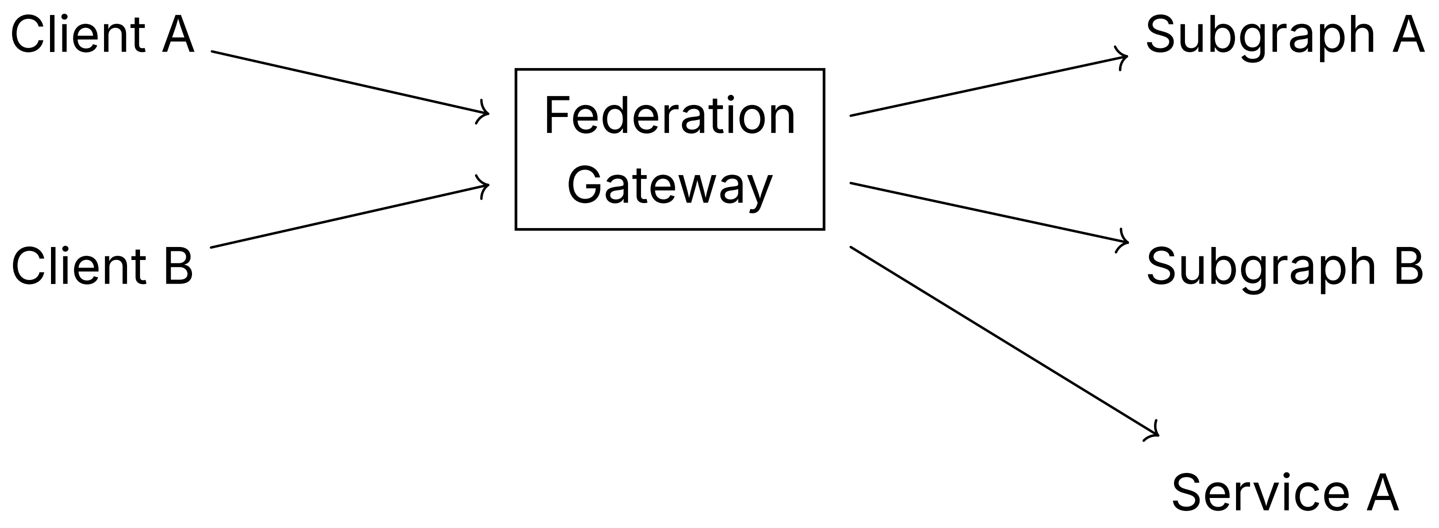
Grafbase

The Federated GraphQL Subscriptions Zoo

Tom Houlé

Federated GraphQL



Subscriptions are special… in GraphQL

> “subscription — a long-lived request that fetches data in response to a sequence of events over time”
>
> — GraphQL spec (Sept 2025)

Subscriptions are special… in GraphQL

> “subscription — a long-lived request that fetches data in response to a sequence of events over time”
>
> — GraphQL spec (Sept 2025)

> “GraphQL supports type name introspection within any selection set in an operation, with the single exception of selections at the root of a subscription operation.”
>
> — GraphQL spec (Sept 2025)

Subscriptions are special… in GraphQL

> “Subscription operations must have exactly one root field.
>
> To enable us to determine this without access to runtime variables, we must forbid the @skip and @include directives in the root selection set.”
>
> — GraphQL spec (Sept 2025)

> “While each subscription must have exactly one root field, a document may contain any number of operations, each of which may contain different root fields. When executed, a document containing multiple subscription operations must provide the operation name as described in GetOperation().”
>
> — GraphQL spec (Sept 2025)

Subscriptions are special… in GraphQL-over-HTTP

Subscriptions are special… in GraphQL-over-HTTP

> “GraphQL Subscriptions are beyond the scope of this specification at this time.”
>
> — GraphQL over HTTP spec (draft)

Subscriptions are special… in GraphQL-over-HTTP

> “GraphQL Subscriptions are beyond the scope of this specification at this time.”
>
> — GraphQL over HTTP spec (draft)

😱

Subscriptions are actually not that special in Federated GraphQL

Subscriptions are actually not that special in Federated GraphQL

Schema of the sales subgraph:

```graphql
type Product @key(fields: "id") {
  id: ID!
}

type Subscription {
  productSales: Product
}
```

Schema of the products subgraph:

```graphql
type Product @key(fields: "id") {
  id: ID!
  name: String!
}

type Query {
  productById(
    id: ID!
  ): Product @lookup
}
```

Subscriptions are actually not that special in Federated GraphQL

Client → Gateway

```graphql
subscription ProductSalesWithName {
  productSales {
    name
  }
}
```

Gateway → sales subgraph

```graphql
subscription {
  productSales {
    id
  }
}
```

Gateway → products subgraph

```graphql
query {
  productById(id: $id) {
    name
  }
}
```

Subscriptions are actually not that special in Federated GraphQL

Data returned to the client:

```
{"name":"Labubu"}
{"name":"Labubu"}
{"name":"Crocs"}
{"name":"Zune"}
{"name":"Furbies (12 pack)"}
{"name":"Labubu"}
{"name": "Google Glass"}
```

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

```
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: graphql-ws, graphql-transport-ws
```

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

```
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: graphql-ws, graphql-transport-ws
```

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

      ```
      Sec-WebSocket-Version: 13
      Sec-WebSocket-Protocol: graphql-ws, graphql-transport-ws
      ```
  - SSE (HTTP/2 and 3)

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

```
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: graphql-ws, graphql-transport-ws
```

  - SSE (HTTP/2 and 3)
  - Multipart

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

      ```
      Sec-WebSocket-Version: 13
      Sec-WebSocket-Protocol: graphql-ws, graphql-transport-ws
      ```
  - SSE (HTTP/2 and 3)
  - Multipart
- **Resource consumption**: one connection between the Gateway and the relevant subgraph per subscribed client, even when they all subscribe to the same events

The problems with Federated Subscriptions

- Lack of transport standardisation has led to **fragmentation**:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

      ```
      Sec-WebSocket-Version: 13
      Sec-WebSocket-Protocol: graphql-ws, graphql-transport-ws
      ```
  - SSE (HTTP/2 and 3)
  - Multipart
- **Resource consumption**: one connection between the Gateway and the relevant subgraph per subscribed client, even when they all subscribe to the same events
- Multi-protocol subscriptions

Multi-protocol subscriptions

- 🖊️ Client — 🍍 → Gateway — 🍎 → 🖊️ Subgraph

Multi-protocol subscriptions

- 🖊️ Client — 🍍 → Gateway — 🍎 → 🖊️ Subgraph
- In the gateway, translations between:
  - SSE,
  - WebSockets
    - `subscriptions-transport-ws`
    - `graphql-ws` / `graphql-transport-ws`
- And different handshake shapes between each!
  - Headers vs websocket init payload shape mismatch

```ts
interface ConnectionInitMessage {
  type: 'connection_init';
  payload?: Record<string, unknown> | null;
}
```

Multi-protocol subscriptions

- 🖊️ Client — 🍍 → Gateway — 🍎 → 🖊️ Subgraph
- In the gateway, translations between:
  - SSE,
  - WebSockets
    - `subscriptions-t`
    - `graphql-ws` / `gra`
- And different hands
  - Headers vs webs

```ts
interface Conne
  type: 'connec
  payload?: Reco
}
```

Alternative: connect the gateway to a message queue

- The idea: the GraphQL federation gateway connects to a message queue (Kafka, NATS, ...), not the subgraphs directly
  - The subgraphs or other services post messages to that queue
- Two implementations
  - Cosmo EDFS
  - Grafbase extensions

Grafbase Extensions

- Pluggable gateway extensions compiled to WebAssembly (WASI preview 2)
  - Define their own directives that will be used by the Gateway for query planning
  - Near-native performance, in-process secure sandbox.
  - Can perform arbitrary IO (but you can restrict that with permissions).
  - Open source extensions from the Grafbase Marketplace or build your own
  - They can act as **virtual subgraphs**

```graphql
extend schema
  @link(
    url: "https://specs.grafbase.com/composite-schemas/v1"
    import: ["@key", "@derive"]
  )
  @link(
    url: "https://extensions.grafbase.com/extensions/nats/0.4.1"
    import: ["@natsPublish", "@natsSubscription"]
  )

input SellProductInput {
  productId: ID!
  price: Int!
}

type Mutation {
  sellProduct(input: SellProductInput!): Boolean!
    @natsPublish(
      subject: "productSales",
      body: { selection: "*" })
}

```

```graphql
type Product @key(fields: "id") {
  id: ID!
}

type ProductSale {
  productId: ID!
  product: Product! @derive
  price: Int!
}

type Subscription {
  sales(subject: String!): ProductSale
    @natsSubscription(
      subject: "{{ args.subject }}"
      selection: "select(.price > 10)"
    )
}
```

Corresponding configuration

```toml
[extensions.nats]
version = "0.4.1"

[[extensions.nats.config.endpoint]]
servers = ["nats://localhost:4222"]
```

Advantages of an extensions-based approach compared to EDFS

- Arbitrary data formats for the messages (not only JSON)
- Customizable and extensible without forking the Gateway. You can write extensions for other pub/sub systems (Kinesis, etc.).
- More powerful filters (`jq` expression language)

Takeaways

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.
- Pros of traditional federated subscriptions
  - **Reuse**: federate existing GraphQL subgraphs, no need to modify them
  - **Control**: subscription fields are managed directly in your own GraphQL subgraph

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.
- Pros of traditional federated subscriptions
  - **Reuse**: federate existing GraphQL subgraphs, no need to modify them
  - **Control**: subscription fields are managed directly in your own GraphQL subgraph
- Pros of subscriptions offloaded to a message queue
  - Stream deduplication
  - Non-GraphQL services can publish to subjects directly
  - Usually **higher performance**, lower memory footprint

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.
- Pros of traditional federated subscriptions
  - **Reuse**: federate existing GraphQL subgraphs, no need to modify them
  - **Control**: subscription fields are managed directly in your own GraphQL subgraph
- Pros of subscriptions offloaded to a message queue
  - Stream deduplication
  - Non-GraphQL services can publish to subjects directly
  - Usually **higher performance**, lower memory footprint

You can mix and match both approaches

Also

Also

Workshop!

Also

Workshop! Tomorrow!

Also

Workshop! Tomorrow!

Grote Zaal - 2nd Floor.

Also

Workshop! Tomorrow!

Grote Zaal - 2nd Floor. 10:45am.

Also

Workshop! Tomorrow!

Grote Zaal - 2nd Floor. 10:45am.

Thank you!

Grafbase

Links

- WebSockets
  - subscriptions-transport-ws
  - Issues and security implications with subscriptions-transport-ws
- SSE
  - GraphQL-SSE spec
- Multipart subscriptions
  - Incremental delivery over HTTP
  - Apollo docs
- Grafbase extensions
- Cosmo EDFS
- Pen Pineapple Apple Pen