Grafbase

The Federated GraphQL Subscriptions Zoo

Tom Houlé

Subscriptions are special… in GraphQL

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

Subscriptions are special… in GraphQL

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

> “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 (draft)

Subscriptions are special… in GraphQL

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 (draft)

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 (draft)

> “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 (draft)

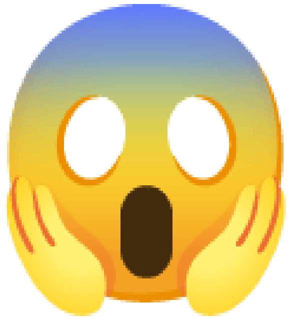
Subscriptions are special… in GraphQL-over-HTTP

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:
  - WebSockets (HTTP/1.1)
    - Subprotocols with protocol negotiation

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

    - Init payloads are not headers
  - SSE (HTTP/2 and 3)
  - Multipart
- 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
- At each step, one of
  - SSE,
  - WebSockets
    - `subscriptions-transport-ws`
    - `graphql-ws` / `graphql-transport-ws`
- And different handshake shapes between each!
  - Headers vs websocket init payload shapes mismatch

Multi-protocol subscriptions

- 🖊️ Client — 🍍 → Gateway — 🍎 → 🖊️ Subgraph
- At each step, one of
  - SSE,
  - WebSockets
    - `subscriptions`
    - `graphql-ws` / `g`
- And different hand
  - Headers vs wel

Event queue to gateway

- The idea: the gateway talks to a message queue (Kafka, NATS, ...), not the subgraphs directly
- Two implementations
  - EDFS
  - Grafbase extensions

EDFS

```graphql
input edfs__NatsStreamConfiguration {
    consumerInactiveThreshold: Int! = 30
    consumerName: String!
    streamName: String!
}

type PublishEventResult {
    success: Boolean!
}

type Query {
    employeeFromEvent(id: ID!): Employee! @edfs__natsRequest(subject: "getEmployee.{{ args.id }}")
}

```

```graphql
input UpdateEmployeeInput {
    name: String
    email: String
}

type Mutation {
    updateEmployee(id: ID!, update: UpdateEmployeeInput!): PublishEventResult!
    @edfs__natsPublish(subject: "updateEmployee.{{ args.id }}")
}

type Subscription {
    employeeUpdated(employeeID: ID!): Employee! @edfs__natsSubscribe(subjects:
    ["employeeUpdated.{{ args.employeeID }}"])
}

type Employee @key(fields: "id", resolvable: false) {
  id: Int! @external
}
```

Grafbase extensions

TODO

Advantages of an extensions-based approach compared to EDFS

- Arbitrary data formats for the messages (not only JSON)
- Customizable and extensible without touching the Gateway. You can write extensions for other pub/sub systems (Kinesis, etc.).
- More powerful filters (`jq` expression language)
- By convention, configuration is usually in your Gateway configuration, not expressed in your subgraph's GraphQL schemas

Takeaways

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.
- Pros of traditional federated subscriptions
  - Federate existing GraphQL subgraphs, no need to modify them
  - Subscription fields are managed directly in your subgraphs, next to your other logic

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.
- Pros of traditional federated subscriptions
  - Federate existing GraphQL subgraphs, no need to modify them
  - Subscription fields are managed directly in your subgraphs, next to your other logic
- Pros of subscriptions offloaded to a message queue
  - Stream deduplication
  - Non-GraphQL services can publish to subjects directly
  - Depends on setup, but usually higher performance with less memory usage

Takeaways

- Federated GraphQL subscriptions require some thinking and planning.
- Pros of traditional federated subscriptions
  - Federate existing GraphQL subgraphs, no need to modify them
  - Subscription fields are managed directly in your subgraphs, next to your other logic
- Pros of subscriptions offloaded to a message queue
  - Stream deduplication
  - Non-GraphQL services can publish to subjects directly
  - Depends on setup, but usually higher performance with less memory usage

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!

Appendices

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