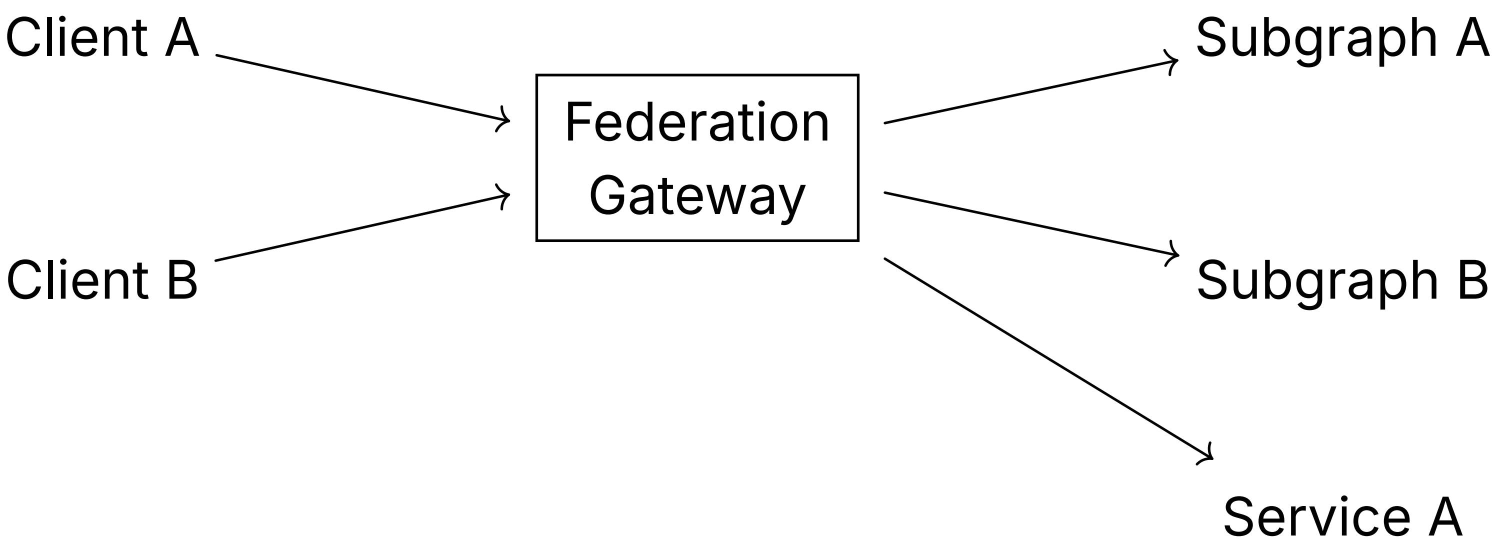
Grafbase

Authorization in Federated GraphQL

Tom Houlé

Federated GraphQL



Why Authorize in the Gateway

- First point of contact to the outside world

Why Authorize in the Gateway

- First point of contact to the outside world
- Whole schema view

Why Authorize in the Gateway

- First point of contact to the outside world
- Whole schema view
- Whole request context

Why Authorize in the Gateway

- First point of contact to the outside world
- Whole schema view
- Whole request context
- Single point of enforcement

Why Authorize in the Gateway

- First point of contact to the outside world
- Whole schema view
- Whole request context
- Single point of enforcement
- Entity resolvers make subgraphs lose context

Entity resolvers make subgraphs lose context

GraphQL

```graphql
query {
 currentUser {
   friends {
     profilePictureUrl
     name
     photos {
       url
     }
   }
 }
}
```

VS

GraphQL

```graphql
query {
  _entities(representations: [
    { __typename: "User", id: "1" }
  ]) {
    ... on User {
      profilePictureUrl
      name
      photos {
          url
      }
    }
  }
}
```

Federation v2 Standard Directives

Federation v2 Standard Directives

- Based on *claims* aka *scopes*
- Claims are derived from:
  - JWT claims
  - Coprocessors

Federation v2 Standard Directives

GraphQL

```graphql
directive @authenticated on
    FIELD_DEFINITION
  | OBJECT
  | INTERFACE
  | SCALAR
  | ENUM
```

Allows accessing the field or type when the request carries *any* verified JWT.

Federation v2 Standard Directives

```GraphQL
directive @requiresScopes(scopes: [[federation__Scope!]!]!) on
      FIELD_DEFINITION
    | OBJECT
    | INTERFACE
    | SCALAR
    | ENUM
```

Allows accessing the field or type when the request has the required scopes/claims.

The outer list wrapper is interpreted as OR. The inner list wrapper is interpreted as AND.

Federation v2 Standard Directives

```graphql
directive @policy(policies: [[federation__Policy!]!]!) on
    FIELD_DEFINITION
  | OBJECT
  | INTERFACE
  | SCALAR
  | ENUM
```

Calls coprocessors or scripts for the given policies. A policy is just a name. The coprocessor has access to claims and context like request headers.

Federation v2 Standard Directives

GraphQL

```graphql
type Query {
    adminDashboard: AdminDashboard
        @policy(policies: [
            ["ip_is_allowlisted"],
            ["is_support_agent", "in_business_hours"],
        ])
}
```

Limitations

- The directives above are sufficient to enable **RBAC** and **limited ABAC**

Limitations

- The directives above are sufficient to enable **RBAC** and **limited ABAC**
- But decisions cannot be tied to **data**
  - Inputs to the fields
  - Output data returned by the subgraphs

Limitations

- The directives above are sufficient to enable **RBAC** and **limited ABAC**
- But decisions cannot be tied to **data**
  - Inputs to the fields
  - Output data returned by the subgraphs
- → **Relationships** cannot be enforced
  - “Users can see the photos on the profile of their friends”
  - “I can see the balance on my own bank account”
  - “I can see the medical records of my own patients”
  - “My direct manager can approve my expense requests if they are < 5000€”

Comprehensive authorization in the Gateway

Comprehensive authorization in the Gateway

- We want to make authorization decisions based on:
  - Request data

GraphQL

```graphql
query {
    user(id: "user_015f91b8-eb7a-418a-8193-f72ddea5760d") {
        socialSecurityNumber
    }
}
```

Comprehensive authorization in the Gateway

- We want to make authorization decisions based on:
  - Request data

```graphql
query {
    user(id: "user_015f91b8-eb7a-418a-8193-f72ddea5760d") {
        socialSecurityNumber
    }
}
```

- And response data too

Comprehensive authorization in the Gateway

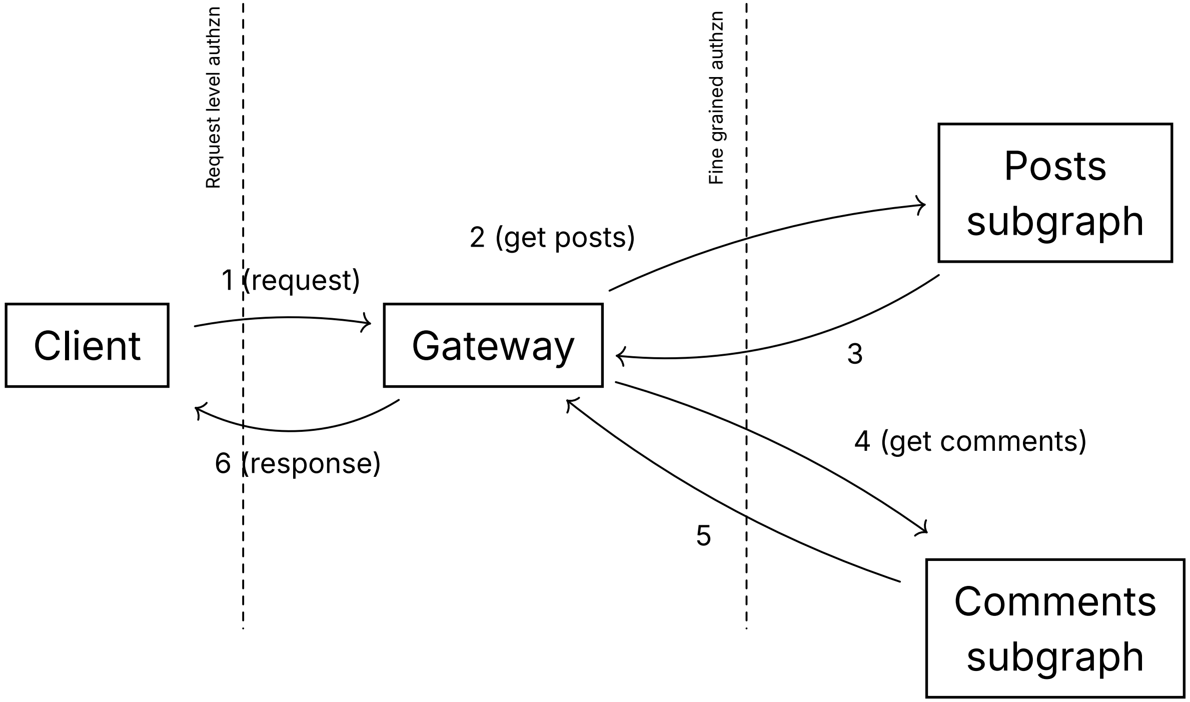
- We want to make authorization decisions based on:
  - Request data

```graphql
query {
    user(id: "user_015f91b8-eb7a-418a-8193-f72ddea5760d") {
        socialSecurityNumber
    }
}
```

- And response data too
- **→ Authorization must be taken into account by the query planner**

Example

```graphql
query PostsWithComments(
    $userID: ID!
) {
  posts(user: $userID) {
    title
    comments(includeHidden: true) {
      author { name }
      commentText
      createdAt
    }
  }
}
```



Our solution

- Achieved with **extensions**.
  - They can define their own directives that will be used by the Gateway for query planning.
  - Compiled to Wasm (WASI preview 2).
    - Near-native performance, in-process secure sandbox.
    - They can perform arbitrary IO (with configurable capabilities).

Pre-subgraph request authorization: define a directive

GraphQL

```graphql
extend schema
  @link(
      url: "https://specs.grafbase.com/grafbase",
      import: ["InputFieldSet"])

directive @authorized(arguments: InputFieldSet = "*")
```

Authorization on input data: apply the directive

GraphQL

```graphql
extend schema
  @link(
      url: "https://extensions.grafbase.com/authorized/0.1.0",
      import: ["@authorized"])

type Query {
    bankAccountByUserEmail(email: String!): BankAccount @authorized
}
```

Authorization on input data: implement authzn logic

Rust

```rust
#[derive(serde::Deserialize)]
struct Authorized<T> {
    arguments: T,
}

#[derive(serde::Deserialize)]
struct BankAccountByUserEmailArguments {
    email: String,
}

fn authorize_query(
    &mut self,
    headers: &mut SubgraphHeaders,
    token: Token,
    elements: QueryElements<'_>,
) -> Result<impl IntoQueryAuthorization, ErrorResponse> {

    let mut builder = AuthorizationDecisions::deny_some_builder();
    for element in elements {
        let DirectiveSite::FieldDefinition(field) = element.directive_site() else {
            unreachable!()
        };
        match (field.parent_type_name(), field.name()) {
            ("Query", "bankAccountByUserEmail") => {
                let authorized: Authorized<BankAccountByUserEmailArguments> = element.directive_arguments()?;
                if authorized.arguments.email != "george@pizzahut.com" {
                    builder.deny(element, "Access denied");
                }
            }
            _ => unreachable!(),
        }
    }

    Ok(builder.build())
}
```

Authorization on output data

- Takes place when a subgraph request is planned
- Will cause the field to become null, with your authorization error in `errors`
- The field and its subfields will not even be requested from the subgraph

Response authorization

GraphQL

```graphql
type User @key(fields: "id") {
  id: ID!
  email: String!
  userType: UserType
  socialSecurityNumber: String @policy(
    policies: [["check_access_to_user_ssn"]]
  )
}
```

Assume we need the `id` and `userType` of the user in addition to the current request context to control access to the social security number.

Response authorization: Problem

Looks good, but...

GraphQL

```graphql
query {
  userByEmail(email: "george@pizzahut.com") {
    socialSecurityNumber
  }
}
```

The `id` and `userType` fields are not going to be available, so our plugin / coprocessor does not have the data it needs to make authorization decisions.

Response authorization: Solution

We define a directive that declaratively pulls in the fields we need in order to make a decision:

GraphQL

```graphql
extend schema
  @link(
      url: "https://specs.grafbase.com/grafbase",
      import: ["FieldSet"])

directive @guard(requires: FieldSet!)
```

Response authorization: Solution

Then we apply it:

GraphQL

```graphql
extend schema
  @link(
      url: "https://extensions.grafbase.com/authorized/0.1.0",
      import: ["@guard"])

type User @key(fields: "id") {
  id: ID!
  email: String!
  userType: UserType
  socialSecurityNumber: String @guard(
    requires: "id userType { canReadSensitiveInfo }"
  )
}
```

Takeaways

- Authorization decision for each annotated field or type can depend on **inputs (arguments)** or **arbitrary associated data**.

Takeaways

- Authorization decision for each annotated field or type can depend on **inputs (arguments)** or **arbitrary associated data**.
- Integrated in the **query planner**
  - It's a requirement
  - Avoids requesting what the current client request is not authorized to see
  - Potentially requests extra fields that are not needed to resolve the GraphQL query, but are required to make authorization decisions.

Takeaways

- Authorization decision for each annotated field or type can depend on **inputs (arguments)** or **arbitrary associated data**.
- Integrated in the **query planner**
  - It’s a requirement
  - Avoids requesting what the current client request is not authorized to see
  - Potentially requests extra fields that are not needed to resolve the GraphQL query, but are required to make authorization decisions.
- All these decisions **batched** by the query planner.

Takeaways

- Authorization decision for each annotated field or type can depend on **inputs (arguments)** or **arbitrary associated data**.
- Integrated in the **query planner**
  - It's a requirement
  - Avoids requesting what the current client request is not authorized to see
  - Potentially requests extra fields that are not needed to resolve the GraphQL query, but are required to make authorization decisions.
- All these decisions **batched** by the query planner.
- Enables **fine grained Attribute-based Access Control (ABAC)** and **Relation-based Access Control (ReBAC)**.

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

- Blog post on fine-grained authorization by Permit.io
- Docs on Apollo Federation v2 built-in authorization directives
- Grafbase Authorization extensions:
  - Grafbase blog post: Custom Authentication and Authorization in GraphQL Federation
  - Example project for authorization extensions