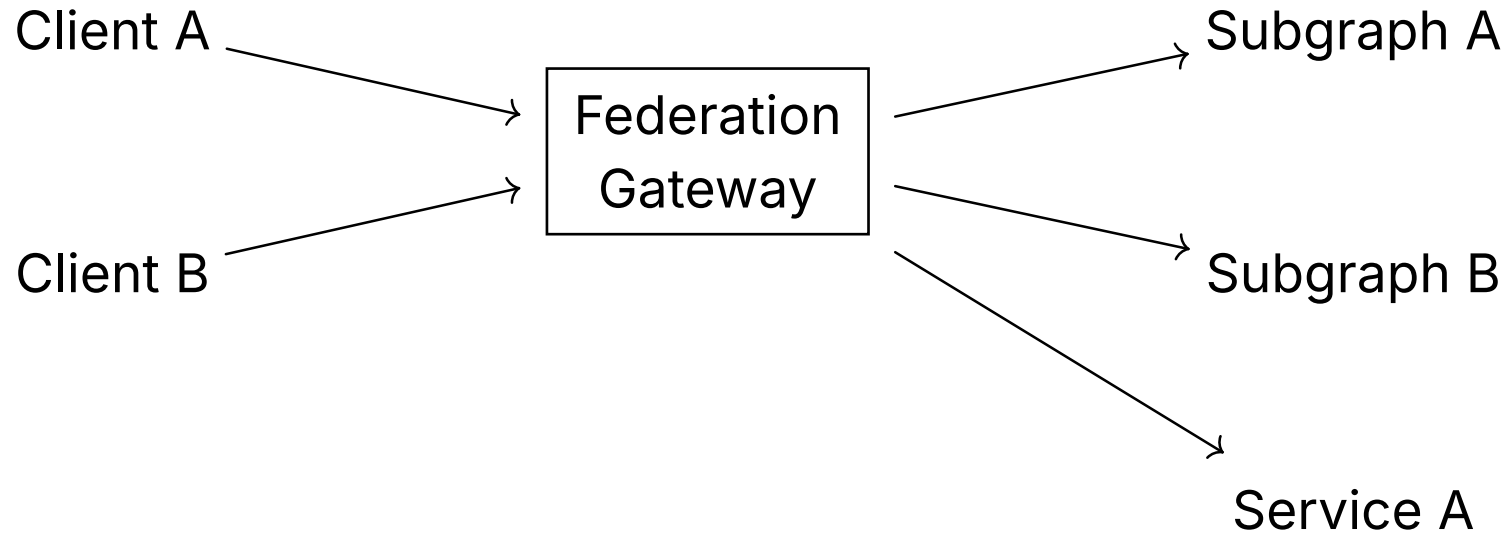




# Authorization in Federated GraphQL

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
- Entity resolvers make subgraphs lose context


# Entity resolvers make subgraphs lose context

```
1 query {  
2   currentUser {  
3     friends {  
4       profilePictureUrl  
5       name  
6       photos {  
7         url  
8       }  
9     }  
10  }  
11 }
```

 GraphQL

VS

```
1 query {  
2   _entities(representations: [  
3     { __typename: "User", id: "1" }  
4   ]) {  
5     ... on User {  
6       profilePictureUrl  
7       name  
8       photos {  
9         url  
10      }  
11    }  
12  }  
13 }
```

 GraphQL

# Federation v2 Standard Directives



# Federation v2 Standard Directives

- They work with *claims*
- Claims are derived from:
  - JWT claims
  - Coprocessors

# Federation v2 Standard Directives


```
1 directive @authenticated on
2     FIELD_DEFINITION
3     | OBJECT
4     | INTERFACE
5     | SCALAR
6     | ENUM
```



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

# Federation v2 Standard Directives

```
1 directive @requiresScopes(scopes: [[federation__Scope!]!]) on
2     FIELD_DEFINITION
3     | OBJECT
4     | INTERFACE
5     | SCALAR
6     | ENUM
```

 GraphQL

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

```
1 directive @policy(policies: [[federation__Policy!]!]) on
2     FIELD_DEFINITION
3     | OBJECT
4     | INTERFACE
5     | SCALAR
6     | 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

```
1 type Query {  
2     adminDashboard: AdminDashboard  
3     @policy(policies: [  
4         ["ip_not_marked_as_potentially_fraudulent"],  
5         ["is_support_agent", "in_business_hours"],  
6     ])  
7 }
```



# Limitations

- The directives above are sufficient to enable RBAC and limited ABAC. Scopes can match roles, or more targeted permissions.

# Limitations

- The directives above are sufficient to enable RBAC and limited ABAC. Scopes can match roles, or more targeted permissions.
- The authorization decisions can however not be tied to the GraphQL query contents
  - The GraphQL document itself: what fields are requested, and at what paths (example: `User.friends`)
  - The data passed in: arguments from literals and variables

# Limitations

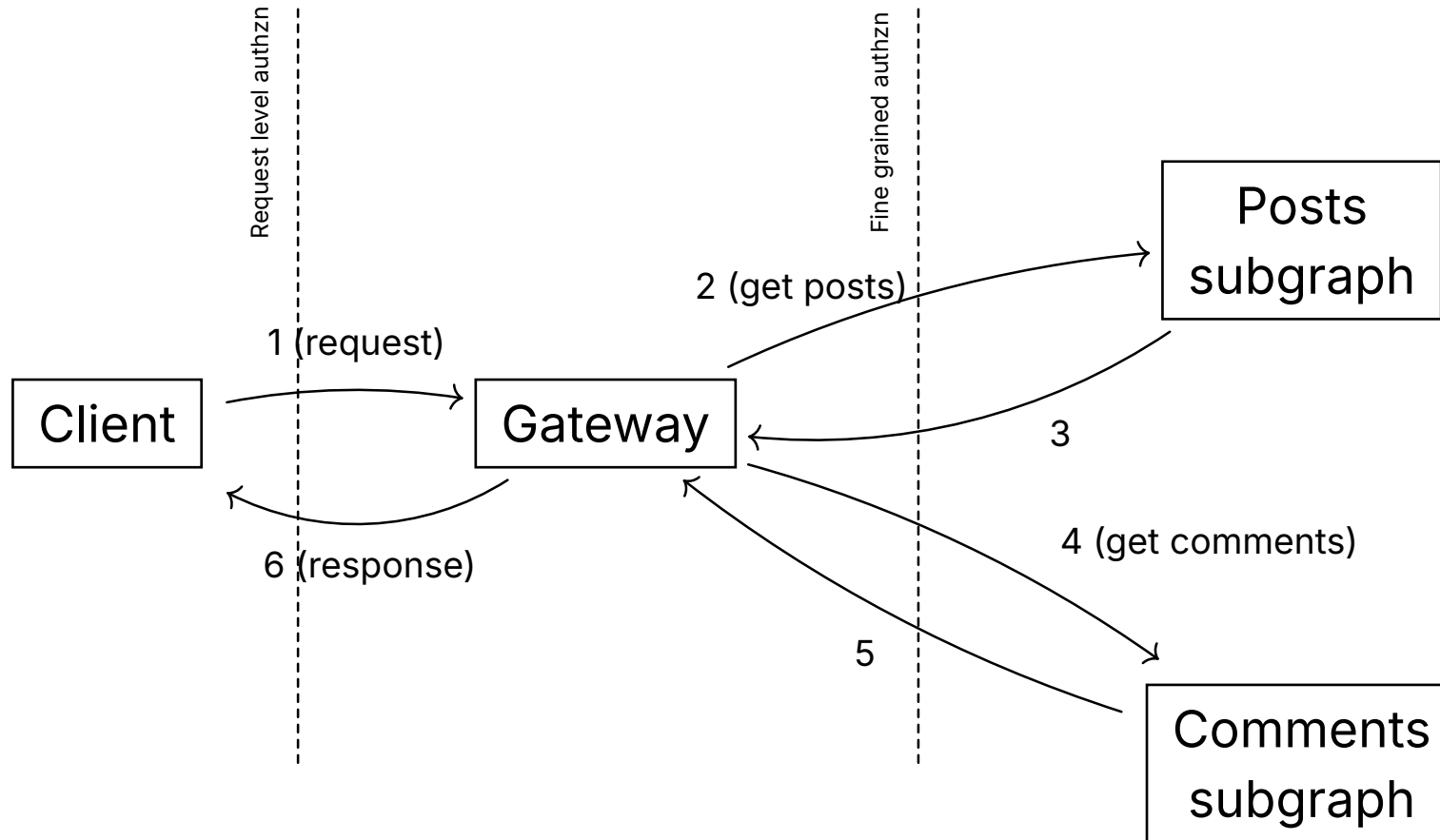
- The directives above are sufficient to enable RBAC and limited ABAC. Scopes can match roles, or more targeted permissions.
- The authorization decisions can however not be tied to the GraphQL query contents
  - The GraphQL document itself: what fields are requested, and at what paths (example: `User.friends`)
  - The data passed in: arguments from literals and variables
- → *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"



# Example

```
1  query PostsWithComments {
2      posts(user: $user) {
3          title
4          comments(includeHidden: true) {
5              author { name }
6              commentText
7              createdAt
8          }
9      }
10 }
```





# Comprehensive authorization in the Gateway

# Comprehensive authorization in the Gateway

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

```
1 query {  
2     user(id: "user_015f91b8-eb7a-418a-8193-f72ddea5760d") {  
3         socialSecurityNumber  
4     }  
5 }
```



# Comprehensive authorization in the Gateway

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

```
1 query {  
2     user(id: "user_015f91b8-eb7a-418a-8193-f72ddea5760d") {  
3         socialSecurityNumber  
4     }  
5 }
```



- And response data too

# Comprehensive authorization in the Gateway

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

```
1 query {  
2     user(id: "user_015f91b8-eb7a-418a-8193-f72ddea5760d") {  
3         socialSecurityNumber  
4     }  
5 }
```



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

## Pre-subgraph request authorization: extensions

- 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 (but you can restrict that with permissions).

## Pre-subgraph request authorization: define a directive

```
1 extend schema
2   @link(
3     url: "https://specs.grafbase.com/grafbase",
4     import: ["InputFieldSet"])
5
6 directive @authorized(arguments: InputFieldSet = "*")
```






## Pre-subgraph request authorization: apply the directive

```
1 extend schema
2   @link(
3     url: "https://extensions.grafbase.com/authorized/0.1.0",
4     import: ["@authorized"])
5
6 type Query {
7   bankAccountByEmail(email: String!): BankAccount @authorized
8 }
```



# Pre-subgraph request authorization: implement authzn logic

```
1  #[derive(serde::Deserialize)]
2  struct Authorized<T> {
3      arguments: T,
4  }
5
6  #[derive(serde::Deserialize)]
7  struct BankAccountByUserEmailArguments {
8      email: String,
9  }
10
11 fn authorize_query(
12     &mut self,
13     headers: &mut SubgraphHeaders,
14     token: Token,
15     elements: QueryElements<'_>,
16 ) -> Result<impl IntoQueryAuthorization, ErrorResponse> {
17
```

 Rust

```

18     let mut builder = AuthorizationDecisions::deny_some_builder();
19     for element in elements {
20         let DirectiveSite::FieldDefinition(field) = element.directive_site() else {
21             unreachable!()
22         };
23         match (field.parent_type_name(), field.name()) {
24             ("Query", "user") => {
25                 let protect: Authorized<BankAccountByUserEmailArguments> = element.directive_arguments()?;
26                 let user_id = protect.arguments.email;
27                 if user_id != "george@pizzahut.com" {
28                     builder.deny(element, "Access denied");
29                 }
30             }
31             _ => unreachable!(),
32         }
33     }
34
35     Ok(builder.build())
36 }

```

## Pre-subgraph request authorization

- 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: take 1

```
1 type User @key(fields: "id") {  
2   id: ID!  
3   email: String!  
4   userType: UserType  
5   socialSecurityNumber: String @policy(  
6     policies: ["check_access_to_user_ssn"]  
7   )  
8 }
```



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: take 1

Looks good, but...

```
1 query {  
2   userByEmail(email: "george@pizzahut.com") {  
3     socialSecurityNumber  
4   }  
5 }
```



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: take 2

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

```
1 extend schema
2   @link(
3     url: "https://specs.grafbase.com/grafbase",
4     import: ["FieldSet"])
5
6 directive @guard(requires: FieldSet!)
```



## Response authorization: take 2

Then we apply it:

```
1  extend schema
2    @link(
3      url: "https://extensions.grafbase.com/authorized/0.1.0",
4      import: ["@guard"])
5
6  type User @key(fields: "id") {
7    id: ID!
8    email: String!
9    userType: UserType
10   socialSecurityNumber: String @guard(
11     requires: "id userType { canReadSensitiveInfo }"
12   )
13 }
```





# Takeaways

- Authorization decision for each annotated field or type can depend on inputs (arguments) or arbitrary associated data on the parent type.
  - Enables fine-grained authorization, for each set of arguments, and each instance type / field / entity

# Takeaways

- Authorization decision for each annotated field or type can depend on inputs (arguments) or arbitrary associated data on the parent type.
  - Enables fine-grained authorization, for each set of arguments, and each instance type / field / entity
- Takes authorization into account for subgraph requests:
  - 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 on the parent type.
  - Enables fine-grained authorization, for each set of arguments, and each instance type / field / entity
- Takes authorization into account for subgraph requests:
  - 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 authorization decisions are batched by the query planner. If you have to call external services to authorize, these calls can be batched as well.

# Takeaways

- Authorization decision for each annotated field or type can depend on inputs (arguments) or arbitrary associated data on the parent type.
  - Enables fine-grained authorization, for each set of arguments, and each instance type / field / entity
- Takes authorization into account for subgraph requests:
  - 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 authorization decisions are batched by the query planner. If you have to call external services to authorize, these calls can be batched as well.
- 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!



# **Appendices**

## Links

- Blog post: Custom Authentication and Authorization in GraphQL Federation
- Example project for authorization extensions