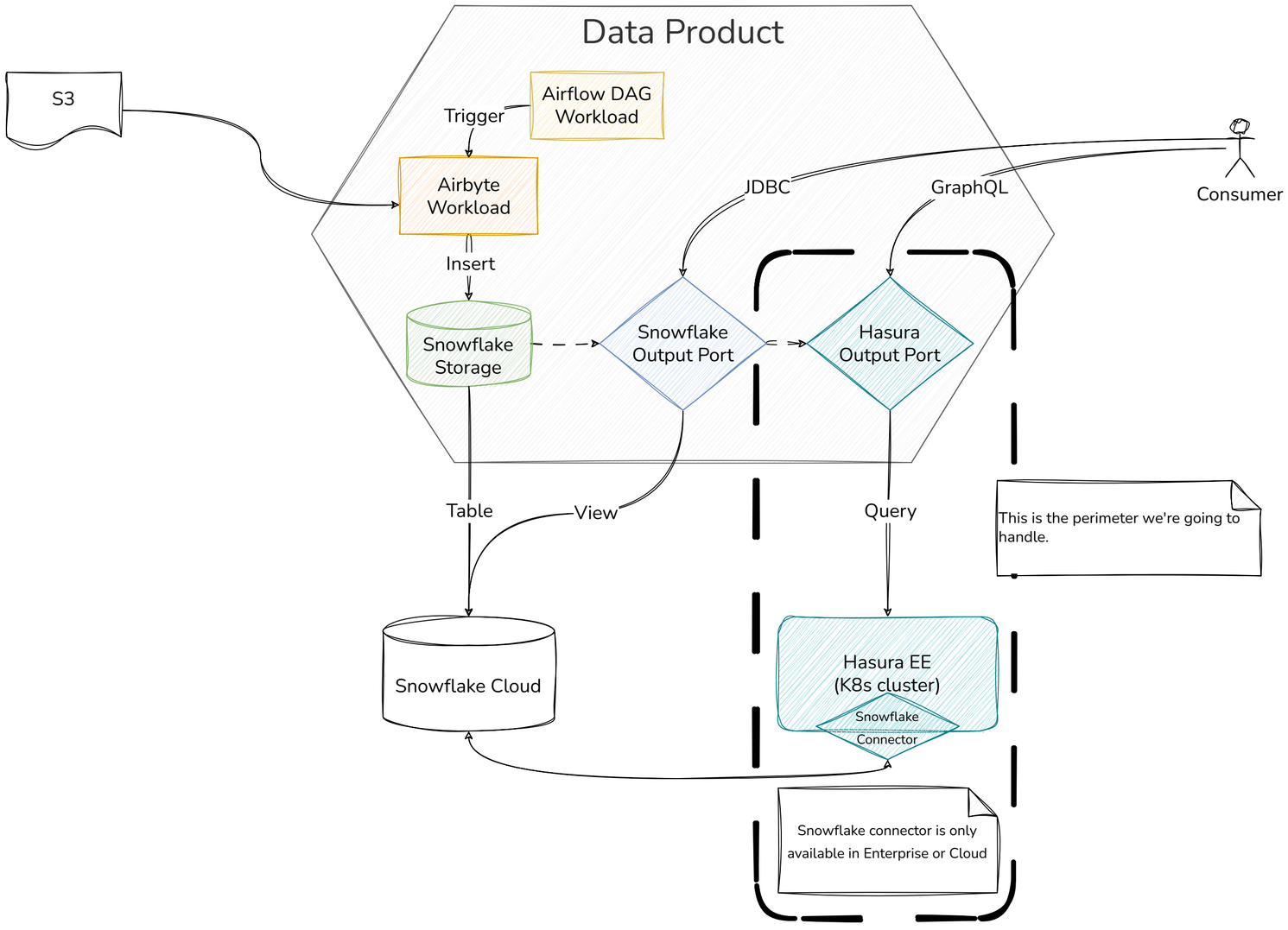

Data Product
S3
Trigger
Airflow DAG
Workload
Airbyte
Workload
Insert
Snowflake
Storage
Snowflake
Output Port
Hasura
Output Port
JDBC
GraphQL
Consumer
Table
View
Query
This is the perimeter we're going to handle.
Snowflake Cloud
Hasura EE
(K8s cluster)
Snowflake
Connector
Snowflake connector is only available in Enterprise or Cloud

DP team

Deploy, update DP/OP

Witboost

Deploy, update DP/OP

Provisioning Coordinator

Provisioning, unprovisioning, ACL updates on Output Ports

Consumer

Access data

Hasura SP

Responsible for provisioning and unprovisioning operations on the Output Ports.

Also services requests to change ACLs on the OP by requesting changes to the role mappings.

Track, untrack, set permissions on tables

Hasura

Data

Table

Data Source

Mapping data:

- roles
- user_roles
- group_roles

Tables:

- roles
- user_roles
- group_roles

PostgreSQL

Authentication requests

Create, delete, update roles and mappings

Webhook/RoleMapper

Responsible for handling authentication webhook calls from Hasura according to role mappings,

Also services requests to create, delete and update role mappings for users and groups.

Create, delete, update roles and mappings

Shown as tables for simplicity; actually GraphQL types accessible via queries & mutations.

DP team

Deploy DP

Witboost

Deploy DP

Provisioning Coordinator

Idempotent, ie does not fail if something it provisions already exists.

1 - Provisioning request for OP

Hasura SP

2 - Retrieve details for component and dependent component (Snowflake OP)

3 - Validate provisioning request

4 - Execute snowflake_track_table request

5 - Execute create_role request

6 - Execute snowflake_create_select_permission request

7 - Generate deploy info

8 - Complete

9 - Send response on polling

4a - Track table

6a - Set permissions

Hasura

Vaccines

roles

VACCINE

Snowflake Cloud (Sandbox account)

roles

PostgreSQL

5c - Mutate

Webhook/RoleMapper

5a - Create role

5b - GraphQL mutate (upsert)

The corresponding unprovisioning operation is very simple: just untrack the table.

- Step 4 changes to an untrack
- Step 5 becomes role delete
- Step 6 disappears

We assume the Data Sources exist already

Validations:

- Check data source exists on Hasura
- Check Snowflake OP is from the same DP
- Check configuration for root fields is valid for GraphQL

```
POST v1/metadata
Content-Type: application/json
X-Hasura-Role: admin
X-Hasura-Admin-Secret: myadminsecretkey
{
  "type": "snowflake_track_table",
  "args": {
    "source": "snowflake",
    "table": "VACCINE",
    "configuration": {
      "custom_name": "Vaccines",
      "custom_root_fields": {
        "select": "Vaccines",
        "select_by_pk": "Vaccine",
        "select_aggregate": "VaccineAgg",
        "insert": "AddVaccines",
        "insert_one":"AddVaccine",
        "update": "UpdateVaccines",
        "update_by_pk": "UpdateVaccine",
        "delete": "DeleteVaccines",
        "delete_by_pk": "DeleteVaccine"
      },
      "comment": "Access to VACCINE table"
    },
    "apollo_federation_config": {
      "enable": "v1"
    }
  }
}
```

```
POST v1/metadata
Content-Type: application/json
X-Hasura-Role: admin
X-Hasura-Admin-Secret: myadminsecretkey
{

 "type":"snowflake_create_select_permission",
   "args":{
      "table":[
         "VACCINE"
      ],
      "role": "dom1.dp1.0.op.readrole",
      "permission":{
         "columns":[

         ],
         "filter":{

         },
         "set":[

         ],
         "allow_aggregations":false
      },
      "source":"snowflake"
   }
}
```

```
POST v1/role
Content-Type: application/json
{
  "role_id":"dom1.dp1.0.op.readrole",
  "component_id": "urn:dmb:cmp:dom1:dp1:0:op",
  "graphql_root_field_name": "dom1_dp1_0_op"
}
```

```
POST v1/graphql
Content-Type: application/json
X-Hasura-Role: admin
X-Hasura-Admin-Secret: myadminsecretket
{
  "query": "mutation InsertRole($component_id:
String = \"\", $graphql_root_field_name:
String = \"\", $role_id: String = \"\") {\n
insert_roles_one(object: {component_id:
$component_id, graphql_root_field_name:
$graphql_root_field_name, role_id: $role_id},
on_conflict: {constraint: roles_pkey,
update_columns: [component_id,
graphql_root_field_name], where:
{graphql_root_field_name: {_eq:
$graphql_root_field_name}, _and:
{component_id: {_eq: $component_id}}}}) {\n
component_id\n    graphql_root_field_name\n
role_id\n  }\n}",
  "variables": {
    "role_id":"dom1.dp1.0.op.readrole",
    "component_id":
"urn:dmb:cmp:dom1:dp1:0:op",
    "graphql_root_field_name": "dom1_dp1_0_op"
  },
  "operationName": "InsertRole"
}
```

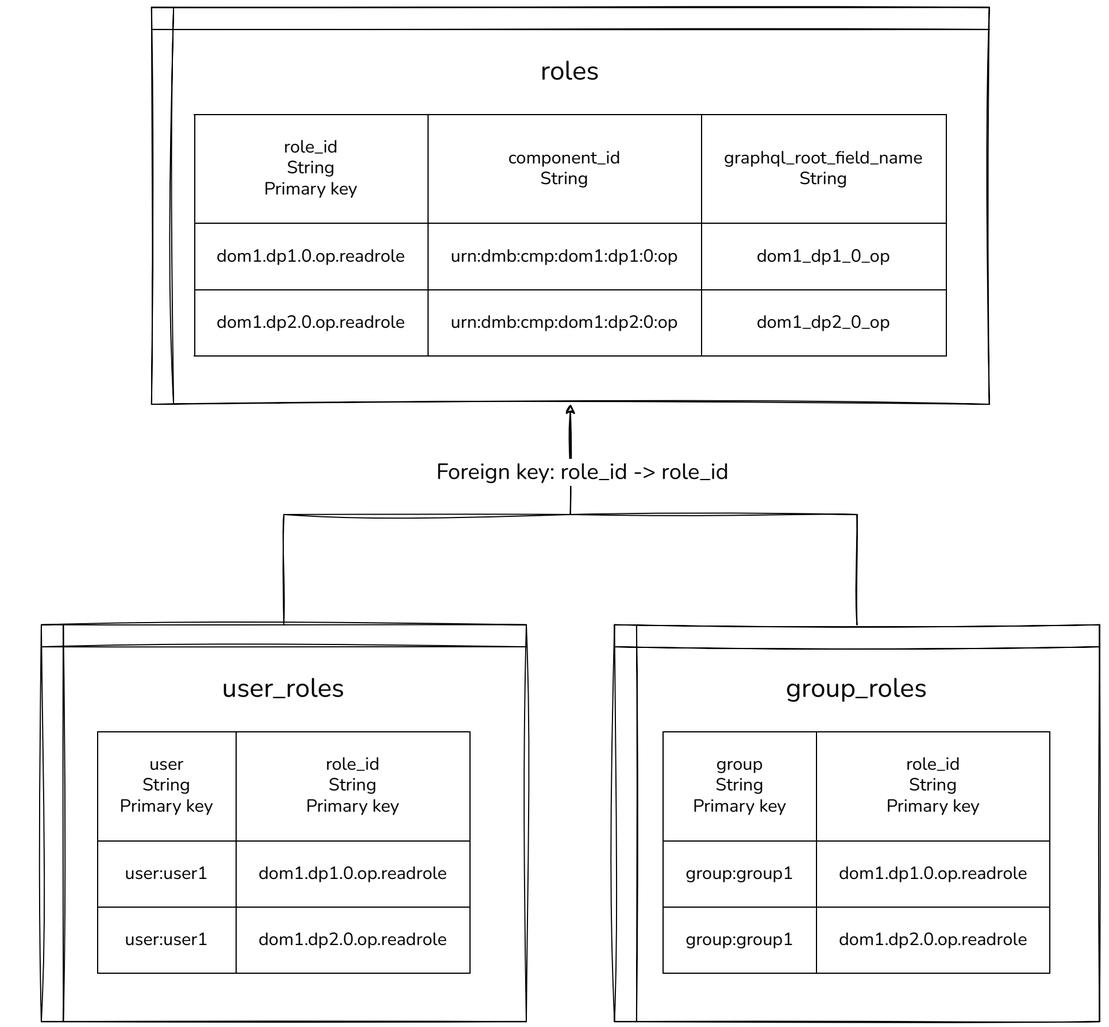
The roles table contains the role id and other information on each role, namely the corresponding component id and GraphQL root field; this is used in sanity checks to ensure multiple components are not using the same role, as well as for retrieving the correct role.

The user_roles & group_roles types map user & groups to sets of roles. The roles for a user are the set union of the user's roles (ie assigned to the user) and group roles (ie assigned to any of the user's groups).

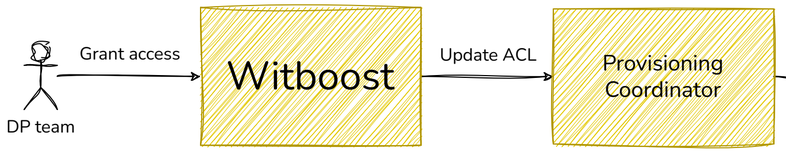
The user and group identifiers in these tables are in the format used by Witboost (ie, user:user1 & group:group1); the translation from the format used by the IdP (and thus used in the JWT) to this format has to be done by the Webhook.

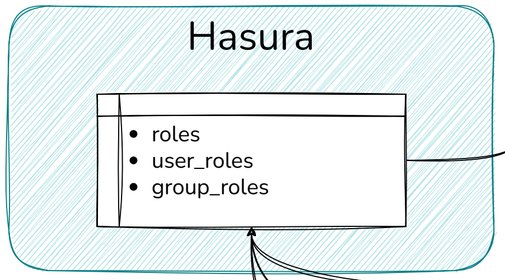
Data is stored in tables in underlying PG database. Permission to read/write is needed only by the role mapping part of the microservice.

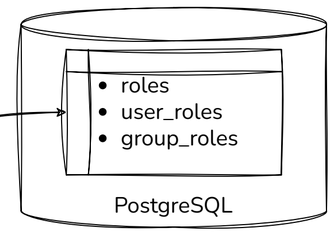
These tables are tracked in Hasura and queried/manipulated via their GraphQL endpoints.

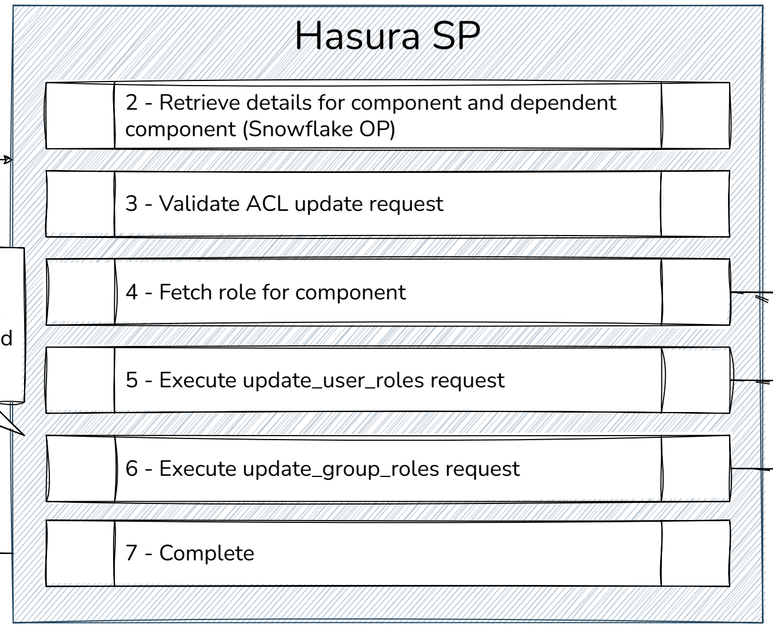


Mapping queries (which query by user/group) are far more frequent than updates (which update by role id), hence the decision to have the user/group as the first PK column, as it makes lookups slighly faster.









1 - ACL Update request

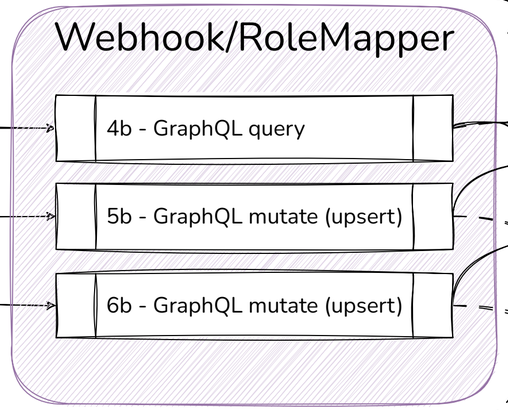
Update means both adding and removing role mappings; handled by the Role Mapper

8 - Send response on polling

4a - Fetch roles

5a - Update user roles

6a - Update group roles



4c - Query roles

5c - Mutate user_roles

6c - Mutate group_roles

We need to use the admin role for these queries, otherwise we end up in an infinte loop. Since we're managing role mappings, we're highly privileged in that sense anyway.

We'll se multiple admin secrets.

Validations:

- Check data source exists on Hasura
- Check Snowflake OP is from the same DP
- Check configuration for root fileds is valid for GraphQL

```
GET
v1/role/component_id/urn:dmb:cmp:dom1:dp1:0:op
```

The provisioner can actually skip this if he knows the role name (eg because it is based on a naming convention).

For users:

```
PUT v1/user_roles
Content-Type: application/json
{
  "role":"dom1.dp1.0.op.readrole",
  "users": ["user:user1", "user:user2"]
}
```

For groups:

```
PUT v1/group_roles
Content-Type: application/json
{
  "role":"dom1.dp1.0.op.readrole",
  "groups": ["group:group1", "group:group2"]
}
```

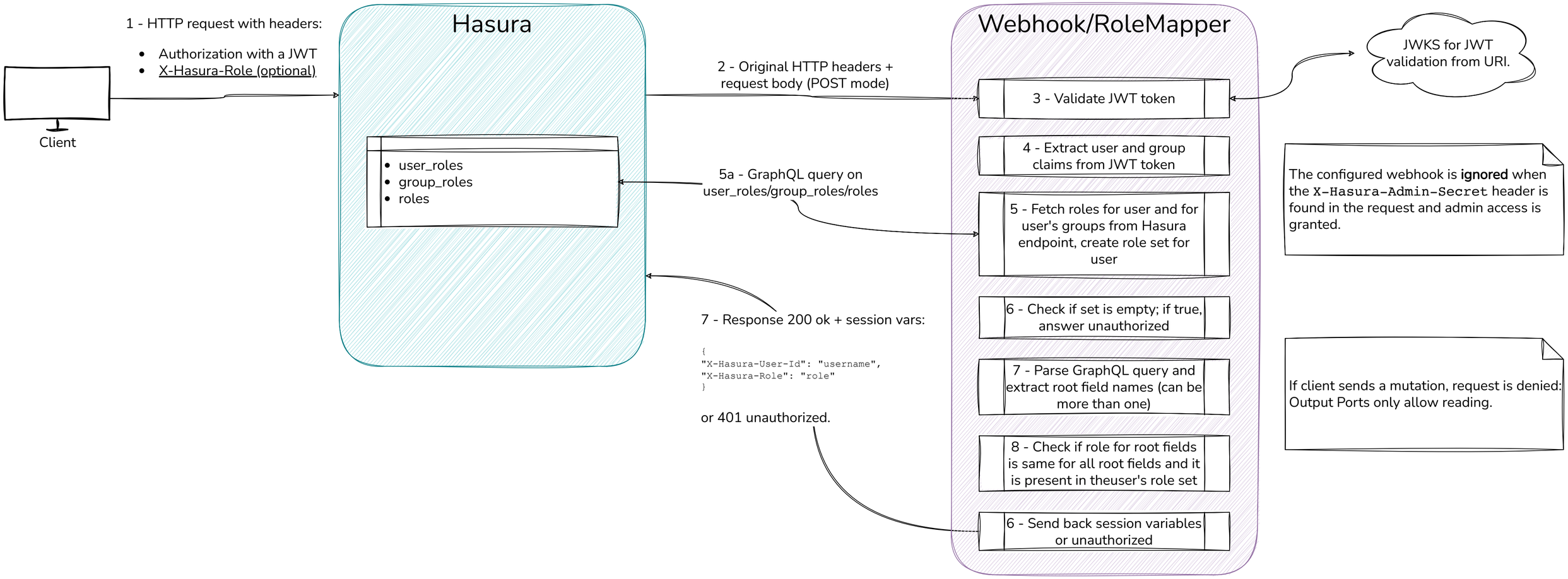
```
POST v1/graphql
Content-Type: application/json
X-Hasura-Role: admin
X-Hasura-Admin-Secret: myadminsecretkey
{
  "query": "query GetRole($component_id:
String) {\n  roles(where: {component_id: {_eq:
$component_id}}) {\n    component_id\n
graphql_root_field_name\n    role_id\n  }\n}",
  "variables": {
    "component_id":
"urn:dmb:cmp:dom1:dp1:0:op"
  },
  "operationName": "GetRole"
}
```

```
POST v1/graphql
Content-Type: application/json
X-Hasura-Role: admin
X-Hasura-Admin-Secret: myadminsecretkey
{
  "query": "mutation UpdateUserRole($user:
String = \"\", $role_id: String = \"\",
$last_update: timestamptz = now) {\n
insert_user_roles(objects: {last_update:
$last_update, role_id: $role_id, user: $user},
on_conflict: {constraint: user_roles_pkey,
update_columns: last_update, where: {}}) {\n
  returning {\n      last_update\n
role_id\n      user\n    }\n  }\n}",
  "variables": {
    "role":"dom1.dp1.0.op.readrole",
    "user": "user:user1"
  },
  "operationName": "UpdateUserRole"
}
```

Above mutation is repeated for every user (can use bulk queries); group updates work the same way, only on group_roles.

If a user & group are not mapped to a role anymore according to the request, we must also remove the previous mapping (omitted to avoid excessive clutter)

We use webhook mode as JWT mode requires customization on the IdP side to add Hasura's custom claims, and with some IdP applying custom logic to generate these claims is hard to do; also it would be too technology specific.



References:

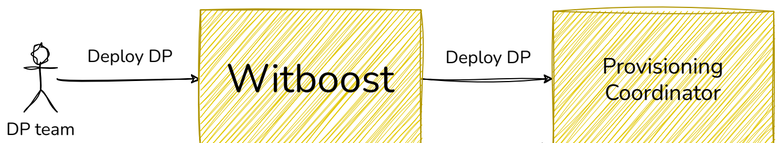
- https://hasura.io/docs/latest/auth/authentication/webhook/
- https://github.com/hasura/graphql-engine/tree/master/community/boilerplates/auth-webhooks

Out of scope for first release

Permissions discussed up to now are based on 1-1 mapping between Output Port and a role. This enables control of data access in an all-or-nothing fashion.

To enable more granular control, multiple roles can be used.

Provisioning (simplified)



Hasura SP

5 - Execute create_role request *for every OP role*

6 - Execute snowflake_create_select_permission request *for every OP role*

Other steps unchanged

```
POST v1/metadata
Content-Type: application/json
X-Hasura-Role: admin
X-Hasura-Admin-Secret: myadminsecretkey
{

 "type":"snowflake_create_select_permission",
   "args":{
      "table":[
         "VACCINE"
      ],
      "role": "dom1.dp1.0.op.readrole",
      "permission":{
         "columns":[
            <column rules for role>
         ],
         "filter":{
            <filter rules for role>
         },
         "set":[

         ],
         "allow_aggregations":false
      },
      "source":"snowflake"
   }
}
```

ACL update (simplified)

Needs customizable access requests; roadmap TBC