World's dumbest clinical data service

- Healthy/Sick
- Authorizations needed stored alongside data, in another table
- Very simple SQL-as-service: resource (Individuals), select, where
- Separate Consents/Authorizations table: allows single individual to be consented to/part of multiple projects (MoHN)

Individuals

id	status
P001	Healthy
P002	Sick
P003	Healthy
P004	Sick
P005	Healthy
P006	Sick

Consents

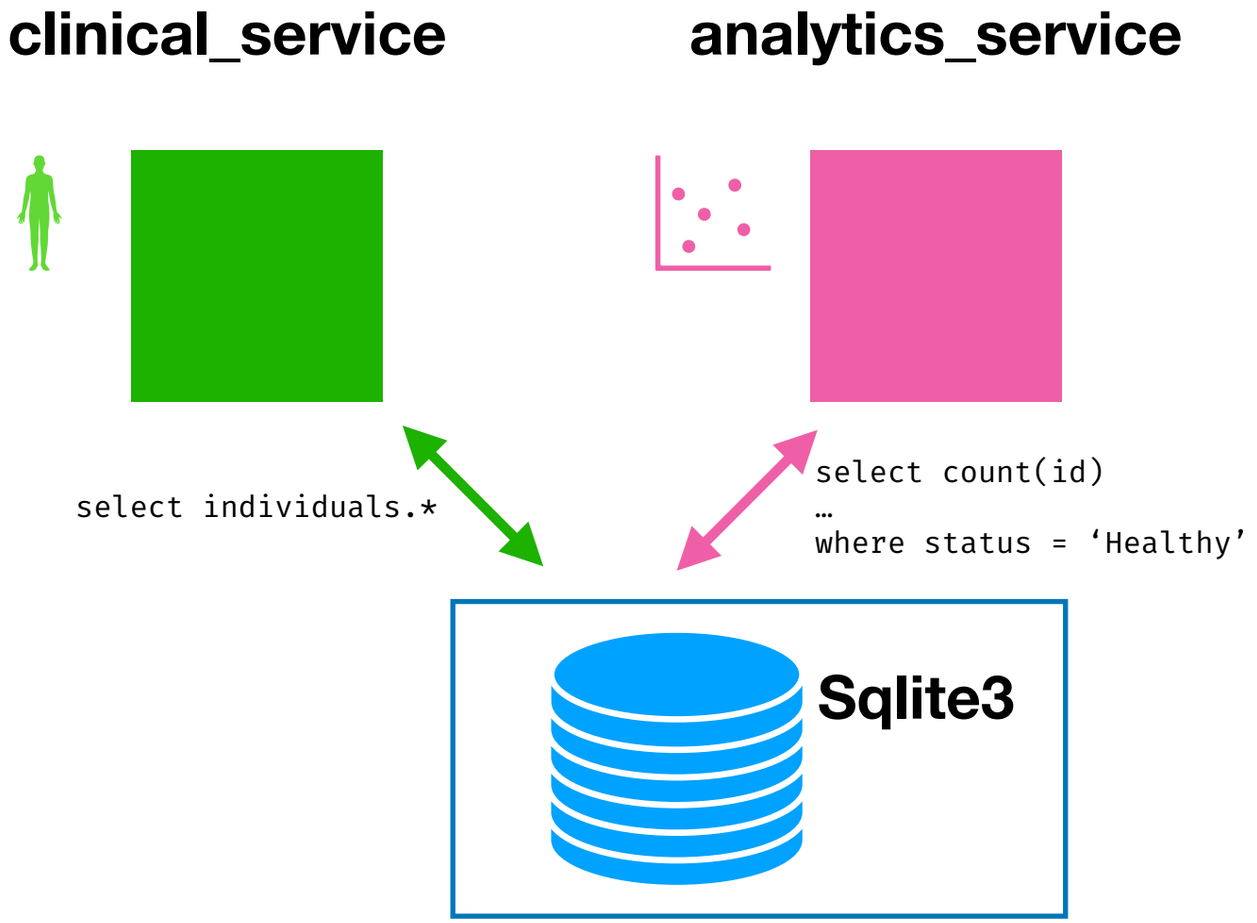
id	project	consent
P001	profyle	TRUE
P002	profyle	TRUE
P003	profyle	TRUE
P004	tf4cn	TRUE
P005	tf4cn	TRUE
P006	tf4cn	TRUE

World's dumbest clinical data service

- Higher level services call the sqlite3 service

  - Clinical service
    - /individuals
    - /individuals/P001

  - Analytics service
    - /n
    - /healthy_fraction



Individuals

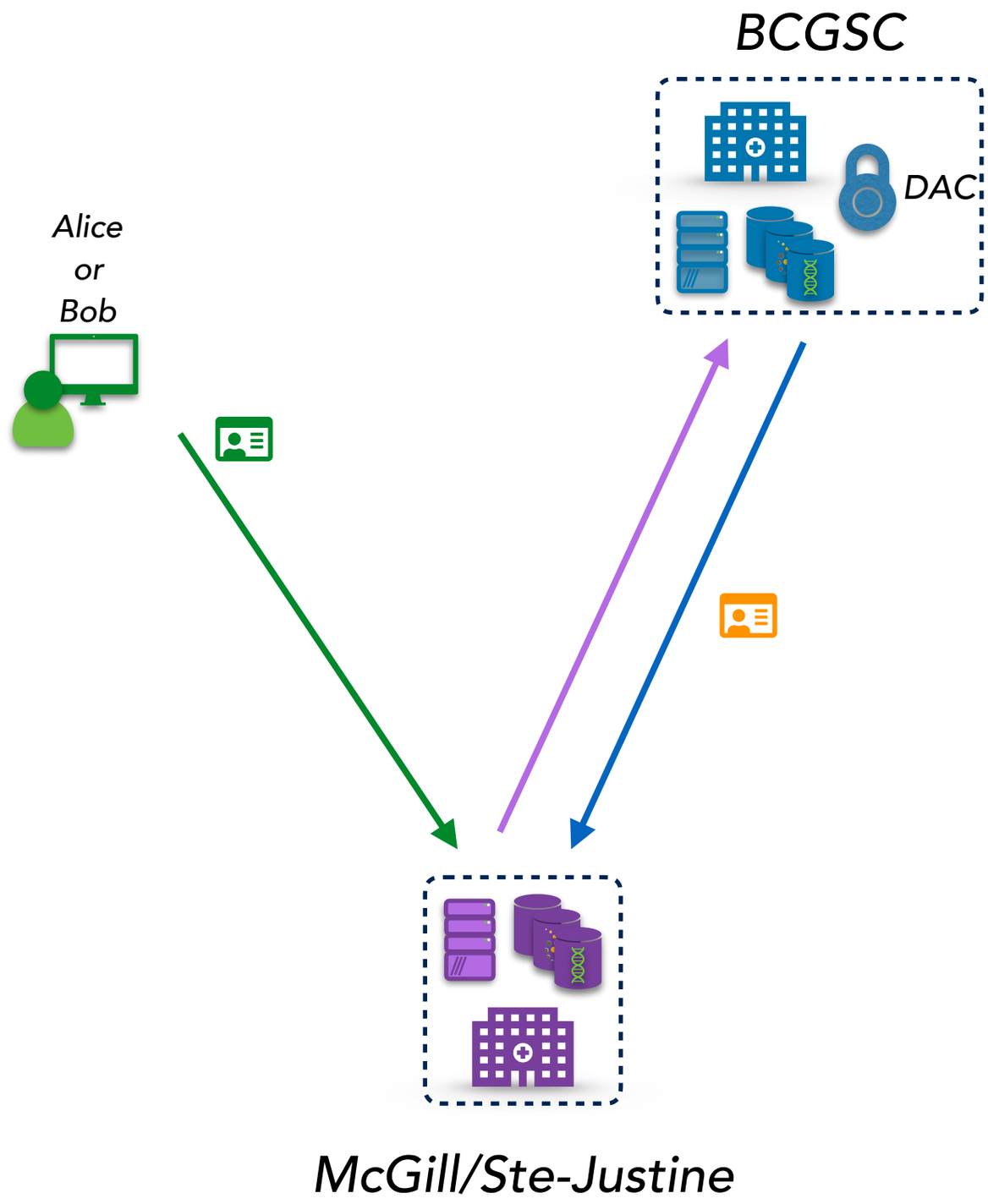
id	status
P001	Healthy
P002	Sick
P003	Healthy
P004	Sick
P005	Healthy
P006	Sick

Consents

id	project	consent
P001	profyle	TRUE
P002	profyle	TRUE
P003	profyle	TRUE
P004	tf4cn	TRUE
P005	tf4cn	TRUE
P006	tf4cn	TRUE

How is request made?

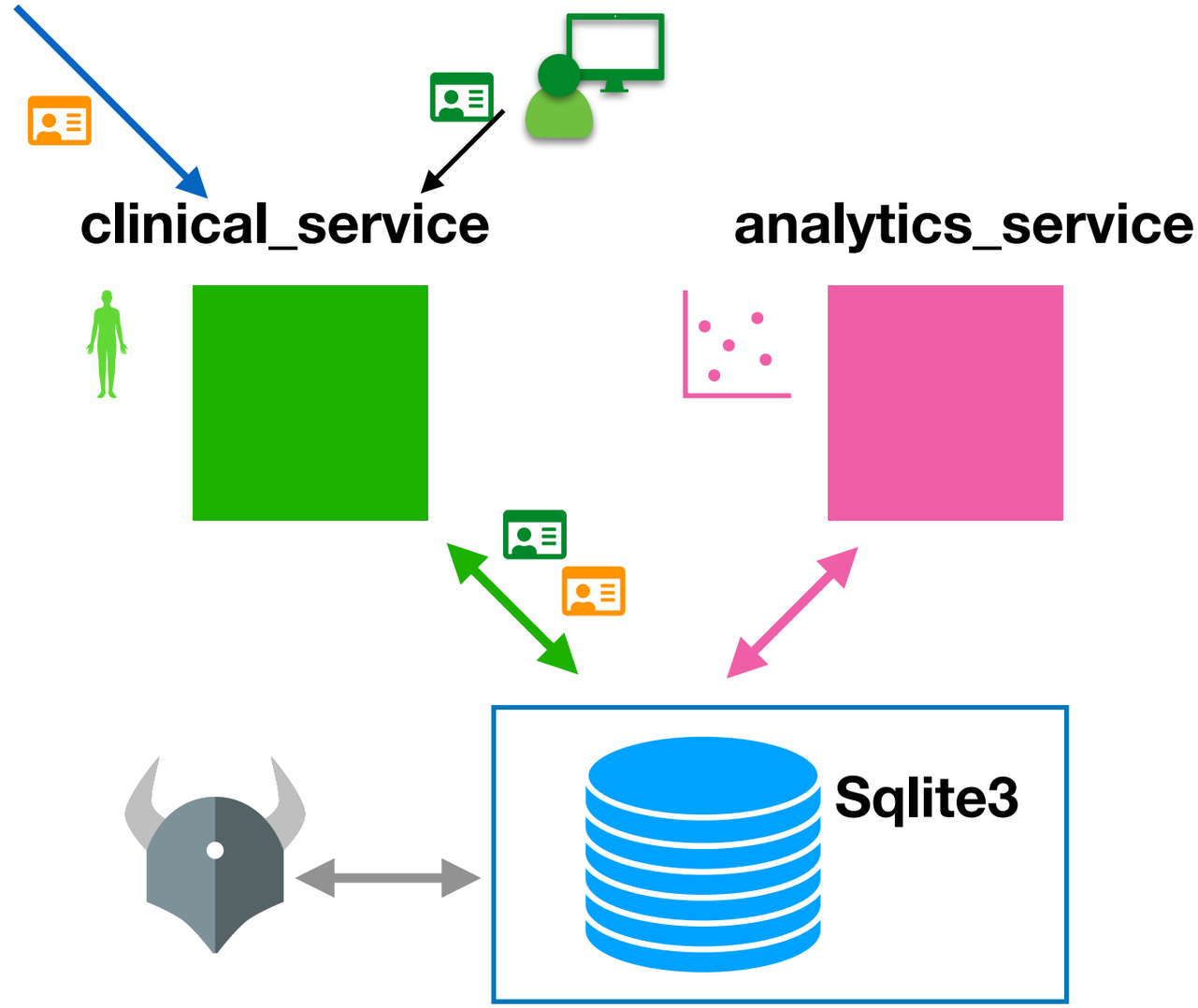
- Request comes in with ID token
- Tyk @ data host requests DAC claims tokens based on that identity from remote DAC portals if they exist
- Not all datasets will have a DAC portal - also support authorization lists



Authorization via OPA

- Tokens (ID token, DAC claims tokens) are passed through the services

- Closest to data, data service queries OPA for authorization info



Individuals

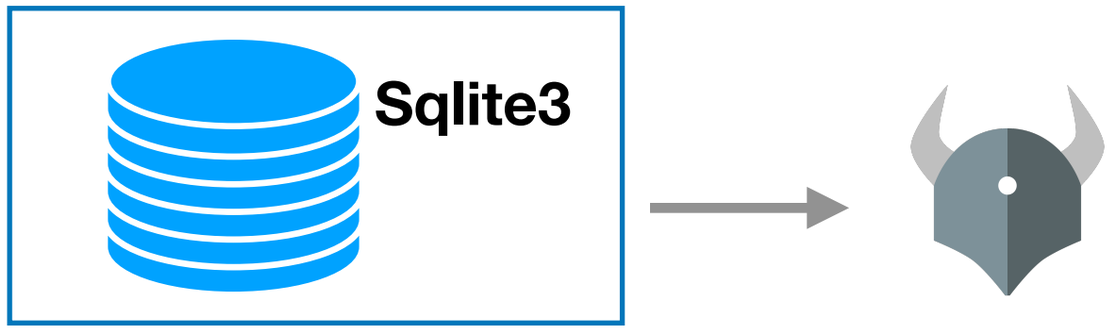
id	status
P001	Healthy
P002	Sick
P003	Healthy
P004	Sick
P005	Healthy
P006	Sick

Consents

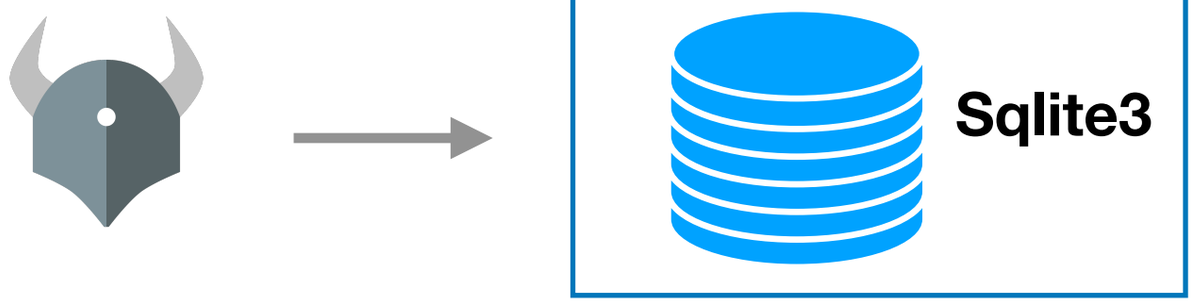
id	project	consent
P001	profyle	TRUE
P002	profyle	TRUE
P003	profyle	TRUE
P004	tf4cn	TRUE
P005	tf4cn	TRUE
P006	tf4cn	TRUE

Authorization via OPA

- Sends query to OPA service
- Service evaluates based on policy, and sends yes/no/“under these conditions” back




```
input = {
  user: 🪪,
  entitlements: [ {‘profyle_member’,🪪}]
  method: “GET”,
  path: [“individuals”],
  unknowns: [“individuals”, “consents”]
}
```



allow = True
or
allow = False
or
allow if....

If any of the tokens are expired, or not signed by right key/modified, authorization fails

```
# get the user (subject) from the identity token
idtoken := {"payload": payload} { io.jwt.decode(input.user, [_, payload, _]) }
subject := idtoken.payload.sub
```

```
default valid_tokens = false
default valid_id_token = false
default valid_entitlement_tokens = false
default allow = false
```

If any of the tokens are expired, or not signed by right key/modified, authorization fails

```
# authorization is denied if the any token is expired or if signature fails validation
# or if entitlement (DAC claim) token doesn't match ID token
# this is also where we'd check that the issuer is one of
# our trusted federation partners, etc.
valid_id_token = true {
  [id_valid, id_header, id_payload] := io.jwt.decode_verify(input.user, {"secret": "secret"})
  id_valid == true
}

# if there are no entitlement tokens, they are valid (there's no invalid ones)
valid_entitlement_tokens = true {
    count(input.entitlements) == 0
}

valid_entitlement_tokens = true {
  # if any of the claims tokens are expired or fail signature validation
  # or the subject isn't the same as the ID subject
  # then we fail
  entitlement_checks := [{"valid":valid, "payload":payload} |
                         [valid, header, payload] := io.jwt.decode_verify(input.entitlements[_].jwt, {"secret": "secret"})]
  all([entitlement_checks[_].valid == true])
  all([entitlement_checks[_].payload.sub == subject])
}

valid_tokens {
    valid_id_token
    valid_entitlement_tokens
}
```

If passes valid_tokens check, and requesting a particular item, succeed if entitlements include consents for that item

```
## access list for projects that don't have a DAC
access_list = {"alice":["tf4cn_member"],
               "bob":  ["tf4cn_member"]}
```

```
# get set of all entitlements from tokens and access list
claim_entitlements := { name | name = input.entitlements[i].name;
                               [ct_head, ct_payload, ct_sig] = io.jwt.decode(input.entitlements[i].jwt);
                               ct_payload[name] == true }
access_list_entitlements := { item | access_list[subject][_] = item }
entitlements := union({claim_entitlements, access_list_entitlements})
```

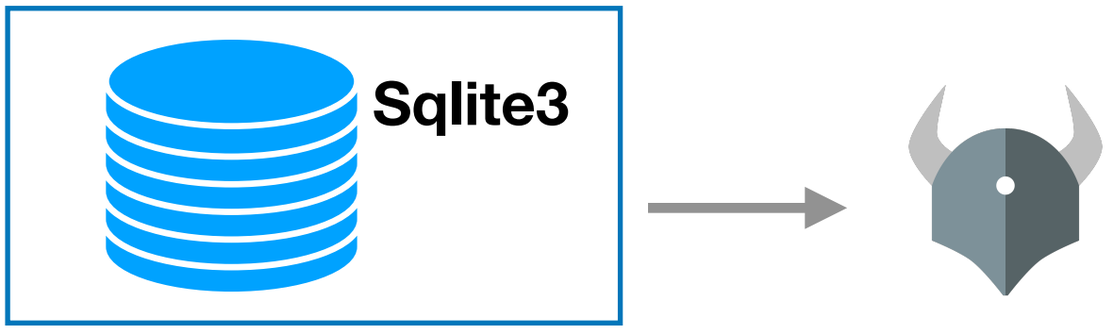
If passes valid_tokens check, and requesting a particular item, succeed if entitlements include consents for that item

```
# authorize a single item
allow = true {
  input.method = "GET"
  input.path = ["individuals", iid]
  # for a single item we can just be given the consents and make the authorization decision
  any({input.consents[i] == entitlements[j]})
}
```

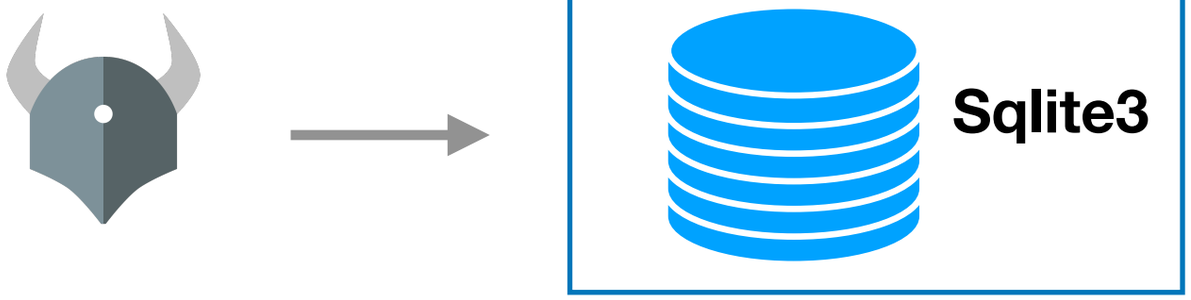
Authorization via OPA

- For valid tokens and a single item, know all relevant information at at query time

- Expect a True/False back



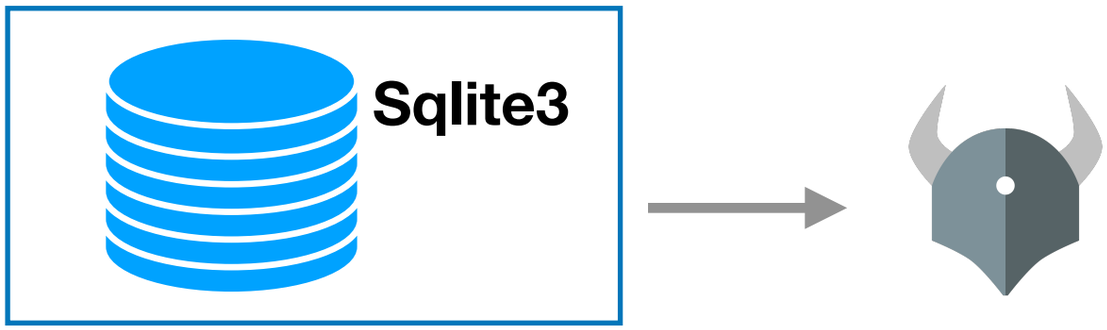

```
input = {
  user: 🪪,
  entitlements: [ {‘profyle_member’,🪪}]
  method: “GET”,
  path: [“individuals”],
  unknowns: [“individuals”, “consents”]
}
```



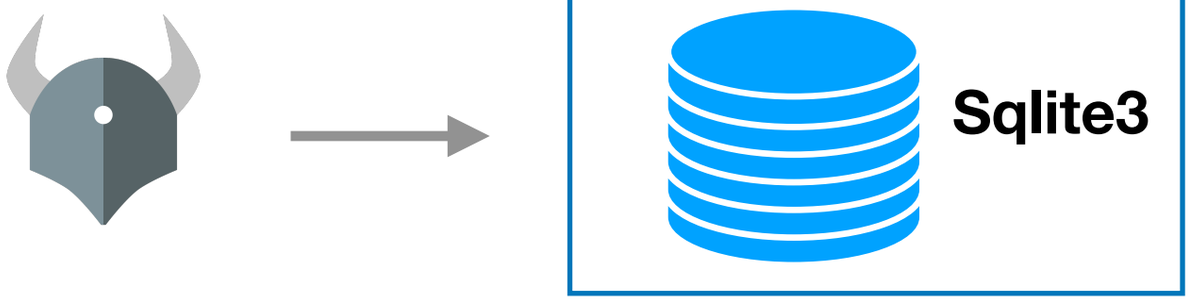
allow = True
or
allow = False
or
allow if....

Authorization via OPA

- For a listing of items, don’t know what items are yet.
- Could:
  - get all items
  - check individually
- Or..



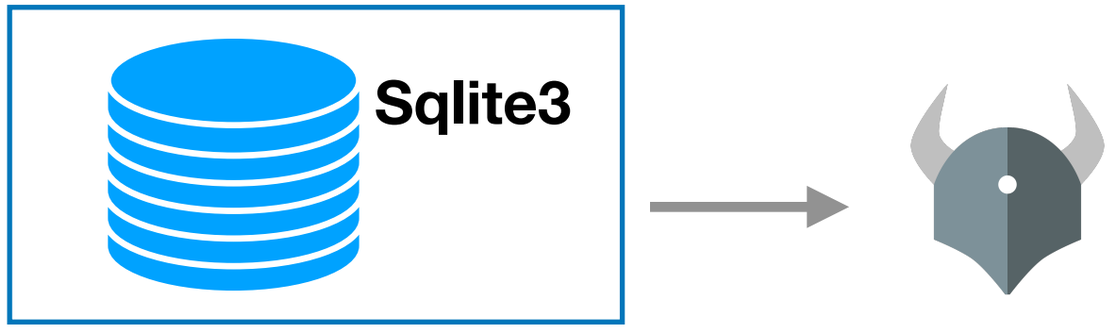

```
input = {
  user: 🪪,
  entitlements: [ {‘profyle_member’,🪪}]
  method: “GET”,
  path: [“individuals”],
  unknowns: [“individuals”, “consents”]
}
```



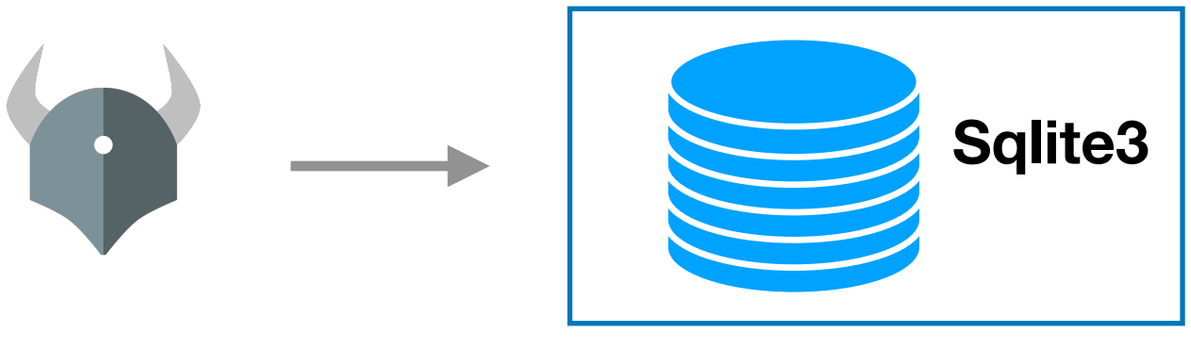
allow = True
or
allow = False
or
allow if....

Authorization via OPA

- If the response is a “yes, if”, is sent as a series of conditions
- Server translates this into SQL clauses
- Clauses are sanitized (by SQLAlchemy) and inserted into query




```
input = {
  user: 🪪,
  entitlements: [ {‘profyle_member’,🪪}]
  method: “GET”,
  path: [“individuals”],
  unknowns: [“individuals”, “consents”]
}
```




```
'INNER JOIN consents
ON (individuals.id = consents.id
      AND consents.consent = true
      AND “profyle_member" =
        consents.project)'
```

A row is allowed if an individual has a consent that matches the DAC entitlements,

```
# authorize items from a list
allow = true {
  input.method = "GET"
  input.path = ["individuals"]

  some x
  row_allowed[x]
}

row_allowed[x] = true {
  data.individuals[x].id = data.consents[x].id
  data.consents[x].consent = true
  proj := data.consents[x].project

  input.entitlements[i].name == proj
  [_, payload, _] := io.jwt.decode(input.entitlements[i].jwt)
  payload[proj] == true
}
```

…or, row is also allowed if an individual has a consent that matches locally-declared entitlements.

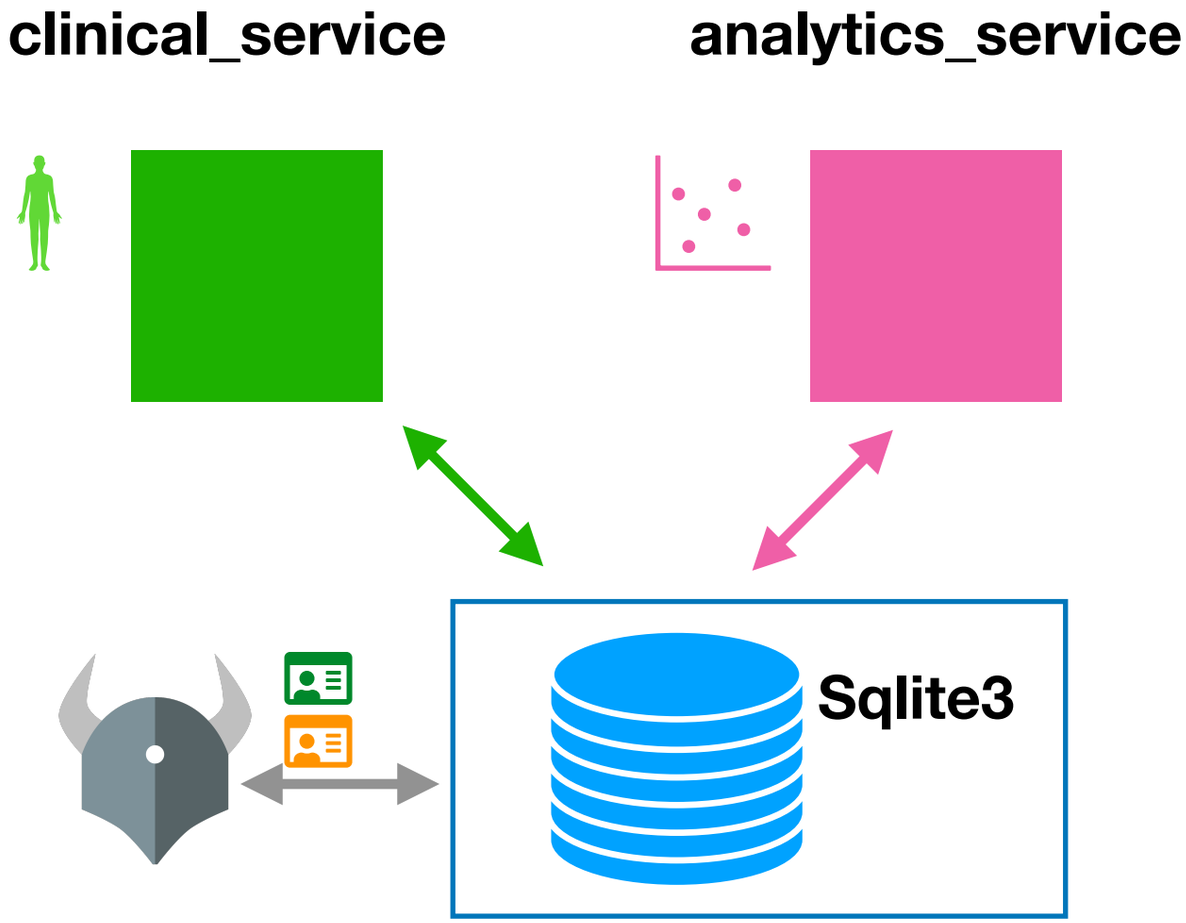
```
## access list for projects that don't have a DAC
access_list = {"alice":["tf4cn_member"],
               "bob":  ["tf4cn_member"]}
```

```
# Alternateily, item is allowed if the data consent matches an entitlement in access list above
row_allowed[x] = true {
  data.individuals[x].id = data.consents[x].id
  data.consents[x].consent = true
  some j
  data.consents[x].project = access_list[subject][j]
}
```

(Have to be a bit more explicit for current SQL translation to work)

Authorization via OPA

- Higher level services call the sqlite3 service
- Don’t have to make any authz checks
  - But probably good idea to check token existence, validity
- Authz enforced by sqlite service following OPA-defined policy
- Unauthorized rows never leave database



Individuals

id	status
P001	Healthy
P002	Sick
P003	Healthy
P004	Sick
P005	Healthy
P006	Sick

Consents

id	project	consent
P001	profyle	TRUE
P002	profyle	TRUE
P003	profyle	TRUE
P004	tf4cn	TRUE
P005	tf4cn	TRUE
P006	tf4cn	TRUE