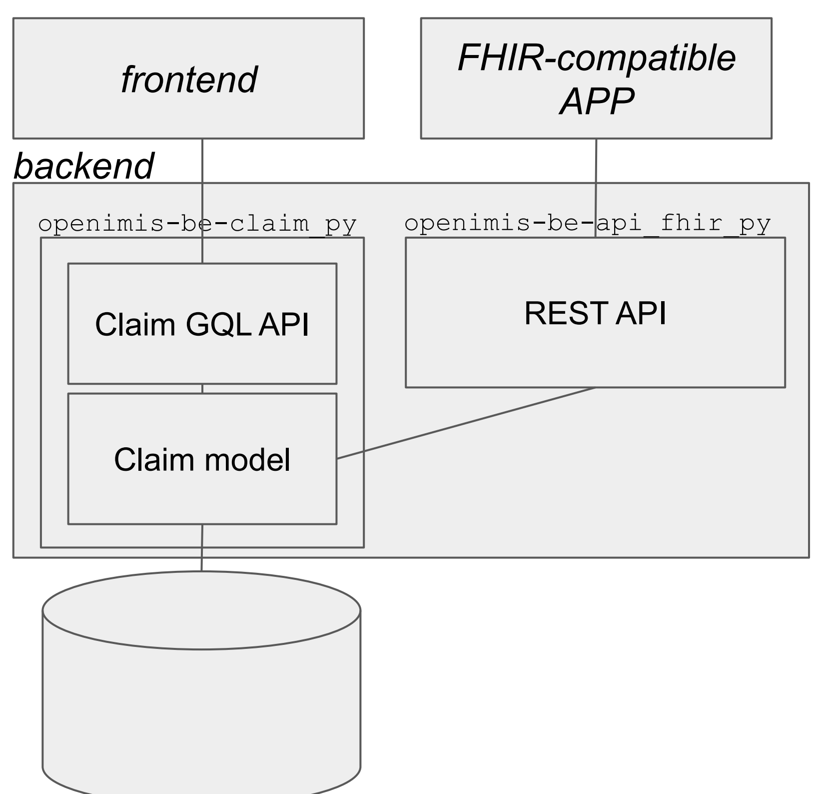
FHIR API
Fine-grained security brainstorming
22/01/2020

Current situation (01/2020)



Current situation:

- FHIR API has been built *before* the Claim module: GQL API *- and the nested security -* was not there at the time
- FHIR API was initially dedicated to (system) **integration** (between *trusted* apps). The security (authentication & authorization) level was not foreseen to be at user level...

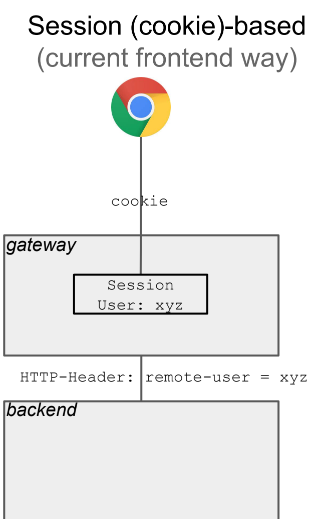
User identity (authentication)

In openIMIS backend, the authentication is based on the REMOTE_USER HTTP header

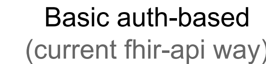
and this is the reason to **NEVER** **expose backend “straight” to outside**

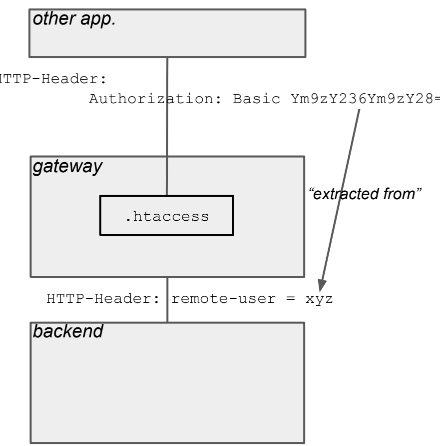
Usage examples:

Session (cookie)-based
(current frontend way)



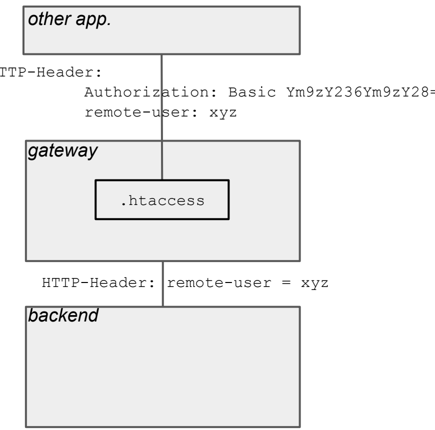
Notes:
- the cookie is set by the legacy openIMIS login
- the session is created ‘intercepting’ the legacy login





Note:
Instead of a basic auth, the external app could be forwarding a SSO token (or X.509 certificate,...) of a (openIMIS) trusted authority

Connection vs. payload identity
(current fhir-api way)



Notes:
- Instead of a HTTP header (visible from outside), we can be transferring identity in message payload (FHIR ‘provenance’ ?)
- The basic-auth can be replaced by 2 way SSL certificates,...

User identity (authentication)

In any cases, the identities (‘logins’) must be agreed upon before the call

...or we can have an ‘**auto-provisioning**’ mechanism: when the request with/for a not yet known user, a user is automatically added to (legacy) openIMIS (in tblUsers).

In openIMIS, there are **‘interactive’** and **‘technical’**:

- **Technical** users are NOT defined in tblUsers and receive their rights ONLY via the (django default) groups/permissions (django rules,...)
- **Interactive** users are defined in tblUsers (legacy openIMIS) table and receive all ‘legacy’ rights from the ‘legacy’ roles/rights (in addition to django groups/permissions)

openIMIS first try to match user as an interactive user

… and “falls back” on a technical user

User identity… extended

Depending on chosen security token format (or agreed user header/payload section), the identity can be 'extended' with some 'profile attributes'(like roles… or even 'the HF the user belongs to',...).

This is especially handy for a fine-grained auto-provisioning (beware of the updates/revoke flows!)

… but requires the various apps (or the token authority) to agree (know) the various attributes (roles,...) used by openIMIS (i.e. tblRoles/tblRights,...).

Objective: completely 'outsource' the user management to an external system (AD, LDAP,...)

Authorization: endpoint security

Every (backend) module:

- can expose any new endpoints (/routes/URLs)... and, in the reference gateway, there is (today) no ‘filtering’ on accessed routes (the gateway only check the access to the openIMIS ‘as a whole’)
- can connect to the database (via the exposed django ‘models’)

As a consequence, by default, every module must perform the authorization part:

- is the user allowed to connect to that endpoint (a.k.a. ‘feature’)?
- can the user access that specific resource (claim,...)?
- ...

Towards central authorization

Provided that we don't "allow" (pure convention) modules to use the other modules' models straight, we could be enforcing a central resource-based authorization

backend

openimis-be-claim_py

Claim GQL API

Claim model

openimis-be-api_fhir_py

REST API

X

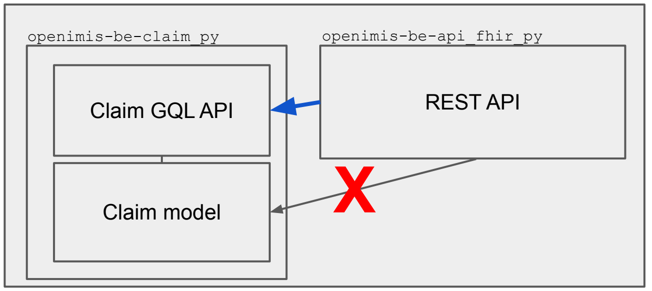
From openimis-be-api_fhir/views.py:

```
77  class ClaimViewSet(BaseFHIRView, mixins.RetrieveModelMixin, mixins.ListModelMixin,
78                     mixins.CreateModelMixin, GenericViewSet):
79      lookup_field = 'uuid'
80      queryset = Claim.objects.all()
81      serializer_class = ClaimSerializer
82
```

→ Uses / depends on

Central authorization alternatives

A. Re-use the fine-grained security implemented in GQL API



Instead of Claim (model) object use ClaimGQLType (that wraps Claim), and [execute GQL queries] (and mutations) straight

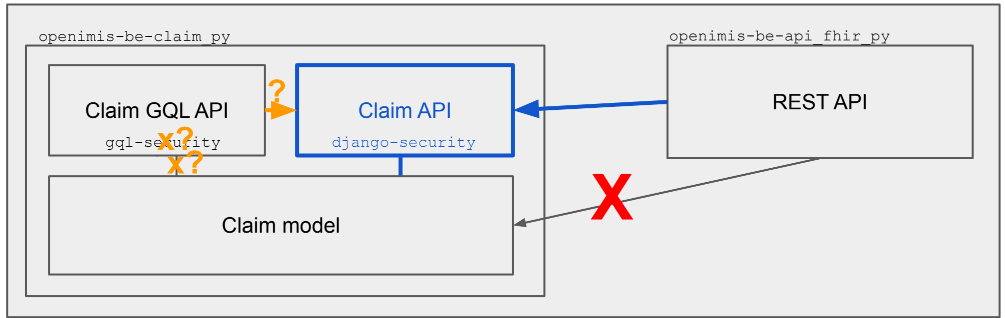
```python
class Claim(core_models.VersionedModel):
    id = models.AutoField(db_column='ClaimID', primary_key=True)
    uuid = models.CharField(db_column='ClaimUUID',
                            max_length=36, default=uuid.uuid4, unique=True)
    category = models.CharField(
        db_column='ClaimCategory', max_length=1, blank=True, null=True)
    insuree = models.ForeignKey(
        insuree_models.Insuree, models.DO_NOTHING, db_column='InsureeID')
    code = models.CharField(db_column='ClaimCode', max_length=8, unique=True)
    date_from = fields.DateField(db_column='DateFrom')
    date_to = fields.DateField(db_column='DateTo', blank=True, null=True)
    status = models.SmallIntegerField(db_column='ClaimStatus')
    adjuster = models.ForeignKey(
        core_models.InteractiveUser, models.DO_NOTHING,
        db_column='Adjuster', blank=True, null=True)
    adjustment = models.TextField(
        db_column='Adjustment', blank=True, null=True)
    claimed = models.DecimalField(
        db_column='Claimed',
        max_digits=18, decimal_places=2, blank=True, null=True)
    approved = models.DecimalField(
        db_column='Approved',
        max_digits=18, decimal_places=2, blank=True, null=True)
    reinsured = models.DecimalField(
        db_column='Reinsured',
        max_digits=18, decimal_places=2, blank=True, null=True)
    valuated = models.DecimalField(
        db_column='Valuated', max_digits=18, decimal_places=2, blank=True, null=True)
    date_claimed = fields.DateField(db_column='DateClaimed')
```

```python
class ClaimGQLType(DjangoObjectType):
    """
    Main element for a Claim. It can contain items and/or services.
    The filters are possible on BatchRun, Insuree, HealthFacility, Admin and ICD in addition to the Claim fields
    themselves.
    """
    attachments_count = graphene.Int()
    client_mutation_id = graphene.String()

    class Meta:
        model = Claim
        exclude_fields = ('row_id',)
        interfaces = (graphene.relay.Node,)
        filter_fields = {
            "uuid": ["exact"],
            "code": ["exact", "istartswith", "icontains", "iexact"],
            "status": ["exact"],
            "date_claimed": ["exact", "lt", "lte", "gt", "gte"],
            "date_from": ["exact", "lt", "lte", "gt", "gte"],
            "date_to": ["exact", "lt", "lte", "gt", "gte"],
```

Central authorization alternatives

B. Implement a ‘service’ (or API) layer in each module (to encapsulate the fine-grained security)...



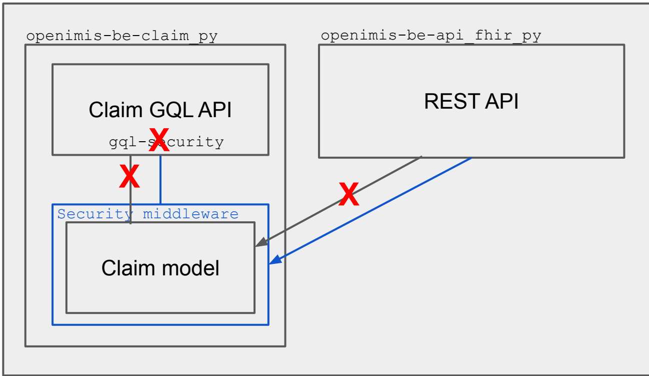
Either:

- we leave Claim GQL as is (and security is duplicated)
- … or we switch Claim GQL to Claim API calls
  but loose the beauty/flexibility of Graphene/django integration (graph ‘navigation” in queries,...)
  ... which is also “bound” to the contribution mechanism in UI (contributed ‘projections’,...)

⟶ Uses / depends on

Central authorization alternatives

C. Move the fine-grained security in a 'middleware', wrapping the model for all modules (and the model's module too)



Alternatives comparison

	Pros	Cons
A. Re-use GQL API	Nothing to change in Claim module. GQL mutations reused (and the associated audit mechanism).	Requires FHIR module (and others) to manipulate GQLTypes (instead of the raw django models), the GQL Mutations (for the create/update/delete),...
B. Service layer as inter-module API	‘Traditional’ approach, API could probably even be exposed ‘straight’ in JSON/XML/…	Requires to design (less flexible) and implement ($-🕖) the service layer Either we change Claim GQL to use the Claim API… or we duplicate security (within Claim module)
C. Wrap django model	Keep current ‘straight-to-model’ flexibility (both for GQL, FHIR… and any other module)	$-🕖 ? Needs to be investigated (existing middleware libs or do we have to build our own, compatibility with django-rules ?,...)