Architecture GeneWeb (Actuelle et Cible)

Date : 11 septembre 2025 Version : master

---

1. Vue d'ensemble

Ce document décrit: - l'architecture actuelle (OCaml) de GeneWeb (exécutables, bibliothèques, flux de données). - une cible Python orientée "queue + events" (CQRS, asynchrone, scalable) avec live editing. - l'utilisation d'une base de données graphe comme read model pour les requêtes de parenté. - un plan de migration incrémental.

Objectifs cibles: - Support de forts volumes (ex: pics potentiellement > 100 req/s) et tâches lourdes isolées. - Time-to-first-byte bas pour lecture, robustesse face aux pics. - Live editing de notes et fiches (WebSocket), gestion des conflits. - Observabilité, sécurité, et conformité RGPD.

---

2. Architecture actuelle (OCaml)

2.1 Composants principaux

- Bibliothèques coeurs: Def, Event, Check, Stats, Mutil, Name, Calendar…
- Geneweb_db: Driver, Collection, Gutil, Database (accès .gwb, personnes/familles, verrous).
- Exécutables:
  - gwd (daemon HTTP/CGI): rendu HTML, auth (wizard/friend), plugins dynamiques, anti-robots, lexiques.
  - gwc (compilation .gw -> .gwo et linkage -> .gwb).
  - gwdiff, gwexport, gwb2ged, ged2gwb, gwgc, etc.
- Plugins: chargement via dynlink (.cmxs) + assets (css/js/lexiques/images).
- Templates/Assets: hd/ (templates), etc/ (scripts/config), images.

2.2 Flux de données

- Ingestion: .gw –(gwcomp)–> .gwo –(db1link)–> .gwb –(check/stats/consang)–> base.
- Service web: gwd lit .gwb, rend HTML, images/assets, auth, notes.
- Export: sélection (gwexport) -> gwb2ged -> fichier GEDCOM.

2.3 Limites

- Ecritures/lectures couplées à un format binaire propriétaire .gwb et verrous fichiers.
- Mise à l'échelle horizontale limitée; rendus HTML server-side.
- Tâches lourdes (imports, consanguinité) non isolées de la voie "lecture".

- Pas de diffusion en temps réel; co-édition complexe.

---

3. Architecture cible (Python, Queue + Events)

3.1 Principes

- CQRS (Command Query Responsibility Segregation): write-side et read-side séparés.
- Event-driven: chaque écriture publie des événements (Kafka/RabbitMQ).
- Asynchrone: workers pour tâches longues (import/export/calc), API allégée.
- Live editing: WebSocket + CRDT/OT pour notes, diffusion des changements.

3.2 Macro-composants

- API Gateway (FastAPI):
  - Endpoints REST/GraphQL (lecture/écriture), WebSocket pour live editing.
  - Auth JWT/OAuth2, rate limiting, validation.
- Write Service (Commands):
  - Validation métier, génération d'événements (person.created, person.updated…).
  - Persistence des agrégats et/ou Event Store (PostgreSQL + table d'événements).
- Bus d'événements: Kafka ou RabbitMQ (topics partitionnés).
- Projectors (Materializers):
  - Consomment les événements et mettent à jour les read models:
    - PostgreSQL (lecture structurée),
    - OpenSearch (recherche texte),
    - Neo4j (graphe de parenté),
    - Redis (cache et pubsub).
- Workers (Celery/RQ):
  - Import GEDCOM, calcul consanguinité, exports, indexing, conversion média.
- Stockage:
  - PostgreSQL (données normalisées, event store),
  - Neo4j (read model graphe),
  - OpenSearch (full-text),
  - Redis (cache, sessions, locks),
  - S3 compatible (images/fichiers).

3.3 Live editing

- WebSocket (FastAPI) + Redis PubSub (ou Kafka bridge) pour diffusion.

- Notes/texte: CRDT (ex: yjs/yrs, automerge) ou OT; versionnage par événements.
- Conflits structurels (liens parents/enfants): commandes synchronisées avec invariants.

3.4 Observabilité et résilience

- Metrics: Prometheus + Grafana; traces OpenTelemetry; erreurs Sentry.
- Retries exponentiels, DLQ (dead letter queue) pour handlers d'événements.
- Circuit breakers (httpx), timeouts, backpressure (queues).

3.5 Sécurité et RGPD

- JWT scopes (wizard, friend, visitor); PII chiffrées au repos (TDE ou app-level).
- RGPD: consent flows, purge/anonymisation, traçabilité des accès.

---

4. Base de données graphe (Read Model)

4.1 Pourquoi un graphe

- Modèle naturel: personnes et familles sont des noeuds; relations parent/enfant, mariage, témoin sont des arêtes.
- Traversées rapides: ascendance/descendance, parenté, chemins, cousins.
- Requêtes expressives: Cypher/Gremlin pour patterns de graphes.
- Evolution du schéma: ajout de nouveaux types de liens sans migration lourde.
- Analytique relationnelle: centralité, communautés, détection d'incohérences.

4.2 Modélisation exemple (Neo4j)

- Noeuds:
  - (Person {id, first_name, surname, birth, death, …})
  - (Family {id, marriage_date, …})
  - (Event {id, type, date, …})
- Arêtes:
  - (p1:Person)-[:PARENT_OF]->(p2:Person)
  - (p1:Person)-[:SPOUSE_OF]-(p2:Person) ou via (Person)-[:IN_FAMILY]->(Family)
  - (w:Person)-[:WITNESS_OF]->(e:Event), (e)-[:ABOUT]->(x)
- Index: contraintes d'unicité sur id; index sur noms normalisés.

4.3 Requêtes types (Cypher)

- Chemin de parenté (borné en longueur):

```
MATCH p=shortestPath((a:Person {id:$idA})-[:PARENT_OF|SPOUSE_OF*..12]-(b:Person {id:$idB}))
RETURN p
```

- Ascendance n générations:

```
MATCH p=(d:Person {id:$id})<-[:PARENT_OF*..$n]-(a:Person)
RETURN a
```

4.4 Intégration CQRS/Event

- Les commandes écrivent dans le write store et publient des événements.
- Un projector graphe consomme les événements et applique les upserts/relations dans Neo4j.
- L'API de lecture oriente parenté utilise Neo4j; les listes/texte passent par PostgreSQL/OpenSearch.

4.5 Limites et précautions

- Ecritures massives: faire via projectors asynchrones (batch ou streaming) pour ne pas bloquer le write path.
- Idempotence: handlers doivent être idempotents; déduplication par clé agrégat + version.
- Dimensionnement: taille mémoire et CPU du cluster graphe; stratégies d'archivage et compaction.

---

5. Diagrammes (ASCII)

5.1 Vue logique

```
Clients
  |-- REST/GraphQL --+--> API Gateway (FastAPI)
  |-- WebSocket -----|
                     +--> Write Service (Commands) --+--> Event Store (Postgres)
                                                     +--> Kafka/RabbitMQ (events)
                                                            |--> Projector Neo4j --> Neo4j
                                                            |--> Projector SQL   --> Postgre
                                                            |--> Projector Search--> OpenSea
                                                            |--> Workers (Celery) --> S3/Exp
                     +--> Read API --> Redis cache --> Postgres/Neo4j/OpenSearch
```

5.2 Séquence écriture

```
Client -> API (POST person.update)
API -> Write Service (commande validée)
Write -> Event Store (append)
Write -> Bus (publish person.updated)
Projectors -> Read models (Neo4j/Postgres/OpenSearch)
```

```
Client <- API (202 Accepted ou 200 selon mode)
```

---

6. Déploiement et scaling (ordre de grandeur)

- API pods: 3-6 (2 vCPU, 4-8 Go), autoscaling sur latence CPU/mémoire.
- Kafka: 3 brokers; 6-12 partitions pour topics critiques.
- Workers: 4-20 (2 vCPU) répartis par files (import/export/compute).
- Postgres: primaire + 1-2 réplicas; pgbouncer.
- Neo4j: cluster 3 noeuds (ou service managé Aura).
- Redis: 3 noeuds (sentinels/cluster) pour cache/pubsub.
- OpenSearch: 3-5 data nodes (selon volume).
- S3: managé (ou MinIO).

SLO cibles (lecture): P95 200-300 ms, erreurs $< 0.5\%$.

---

7. Plan de migration

1) Interop OCaml: encapsuler lecture .gwb/exports en service RPC; exposer API Python devant.
2) Projections: batch et streaming d'événements pour alimenter Postgres/Neo4j/OpenSearch.
3) Ecriture: introduire Write Service Python; conserver lecture sur projections.
4) Workers: déplacer import GEDCOM, consanguinité, exports vers Celery; instrumenter.
5) Live: activer WebSocket + CRDT pour notes; ensuite édition structurelle avec commandes.
6) Optionnel: retrait progressif du .gwb, ou maintien comme format d'échange/compat.

---

8. Risques et mitigations

- Incohérences cross-model: idempotence, versionnage, replays; contrats d'événements.
- Coût opérationnel multi-stores: automatiser IaC, observabilité, sauvegardes.
- Conflits d'édition: CRDT pour notes, workflow de validation pour structure.
- RGPD: masquage sélectif, censure en projection, journalisation des accès.

---

9. Conclusion

L'architecture "queue + events" avec read models spécialisés (SQL, OpenSearch, Graphe) apporte: - scalabilité lecture/écriture et isolation des traitements lourds, - expressivité et performance pour les requêtes généalogiques (graphe), - temps réel (live editing) et robustesse aux pics.

La migration incrémentale permet de conserver la valeur OCaml existante jusqu'à la bascule complète.