Dossier projet - Annexes

GeoChallenge Tracker

Jean Ceugniet

Table des matières

1 Lexique du géocaching (notions utiles au projet)

Afin de faciliter la lecture du dossier par des lecteurs non familiers du géocaching, je vous propose ci-après un petit lexique. Il rassemble uniquement les notions indispensables à la compréhension du projet *GeoChallenge Tracker*, en donnant une définition claire des termes utilisés dans le document.

- **Cache** : conteneur physique dissimulé par un joueur, localisé par ses coordonnées GPS et listé sur une plate-forme de géocaching.
- **Cache trouvée** : cache qui a été effectivement localisée et loguée comme trouvée par un joueur. Ces données constituent la base de calcul pour évaluer la progression dans les challenges.
- **Géocacheur** : joueur pratiquant le géocaching, qui recherche des caches posées par d'autres et en publie éventuellement lui-même.
- **Géocaching** : chasse au trésor moderne qui utilise un GPS ou un smartphone pour localiser des contenants cachés (appelés "géocaches") dissimulés partout dans le monde par d'autres participants. Une fois trouvée, on signe le carnet de bord à l'intérieur pour prouver la trouvaille.
- **Owner / Poseur** : géocacheur ayant créé et publié une cache. Son rôle est de la maintenir en bon état et de gérer son descriptif en ligne.
- **Found it** : log positif indiquant qu'un joueur a trouvé la cache.
- **DNF (Did Not Find)** : log indiquant qu'un joueur a cherché la cache mais ne l'a pas trouvée.
- **Challenge Cache** : type particulier de cache qui ne peut être loguée comme trouvée que si le joueur a fait un *found it* **ET** a rempli certaines conditions prédéfinies (ex. avoir trouvé 100 caches d'un type donné).
- **Challenge (au sens du projet)** : défi géocaching, correspondant aux conditions fixées par une *challenge cache*. Dans *GeoChallenge Tracker*, les challenges sont importés depuis les caches détectées comme "challenge", puis suivis individuellement par chaque joueur.
- **Task (tâche de challenge)** : sous-condition définie par l'utilisateur pour un challenge donné, permettant de préciser la manière dont il souhaite interpréter ou suivre ce défi.
- **Progression** : enregistrement des étapes franchies par un joueur dans la réalisation d'un challenge, calculée à partir des caches trouvées et des tâches associées.
- **Target (cible)** : cache identifiée par l'application comme potentiellement utile pour compléter un challenge en cours, en fonction des conditions définies dans ses tâches.
- **D/T (Difficulté/Terrain)** : double cotation officielle indiquant la difficulté intellectuelle (D) et la difficulté physique/terrain (T) d'une cache. Ces valeurs peuvent être utilisées comme conditions dans les challenges.
- **Type (de cache)** : il existe différentes catégories de caches nécessitant des actions différentes : traditionnelles, mystères, virtuelles…
- **Taille (de cache)** : indique la taille physique de la cache, allant de micro à large.
- **Attribut (de cache)** : étiquette descriptive rattachée à une cache (ex. disponible la nuit, accessible aux enfants). Dans le projet, certains attributs servent de critères pour les tâches de challenge.
- **Marker clustering** : regroupement de points sur une cache interactive
- **Grammaire AST** : représentation d'expressions ou conditions sous forme d'arbre syntaxique abstrait, permettant leur interprétation par le logiciel.

2 Maquettes

2.1 Menu

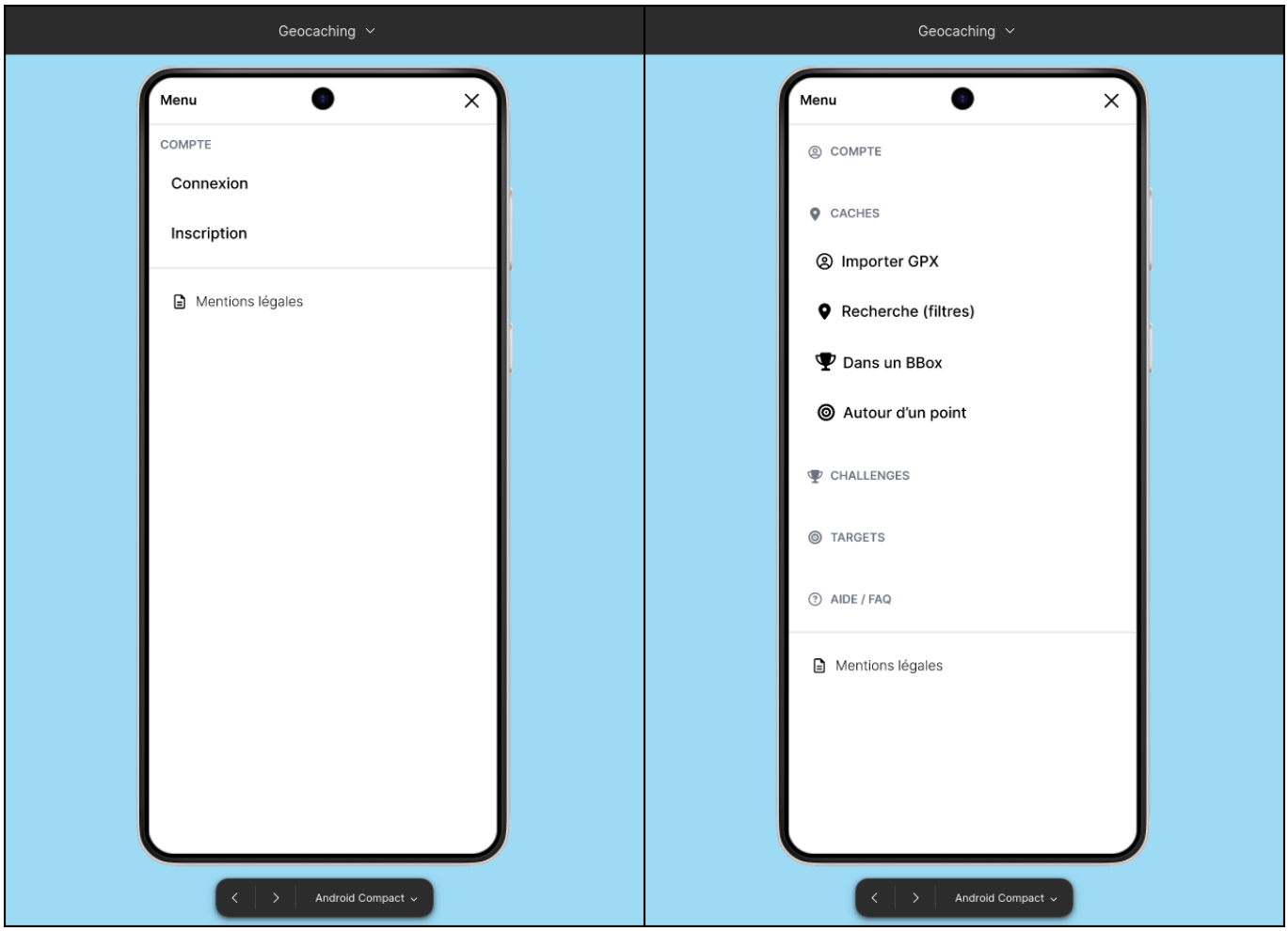


Figure 1 : Menu non loggé / loggé

2.2 Page d’accueil

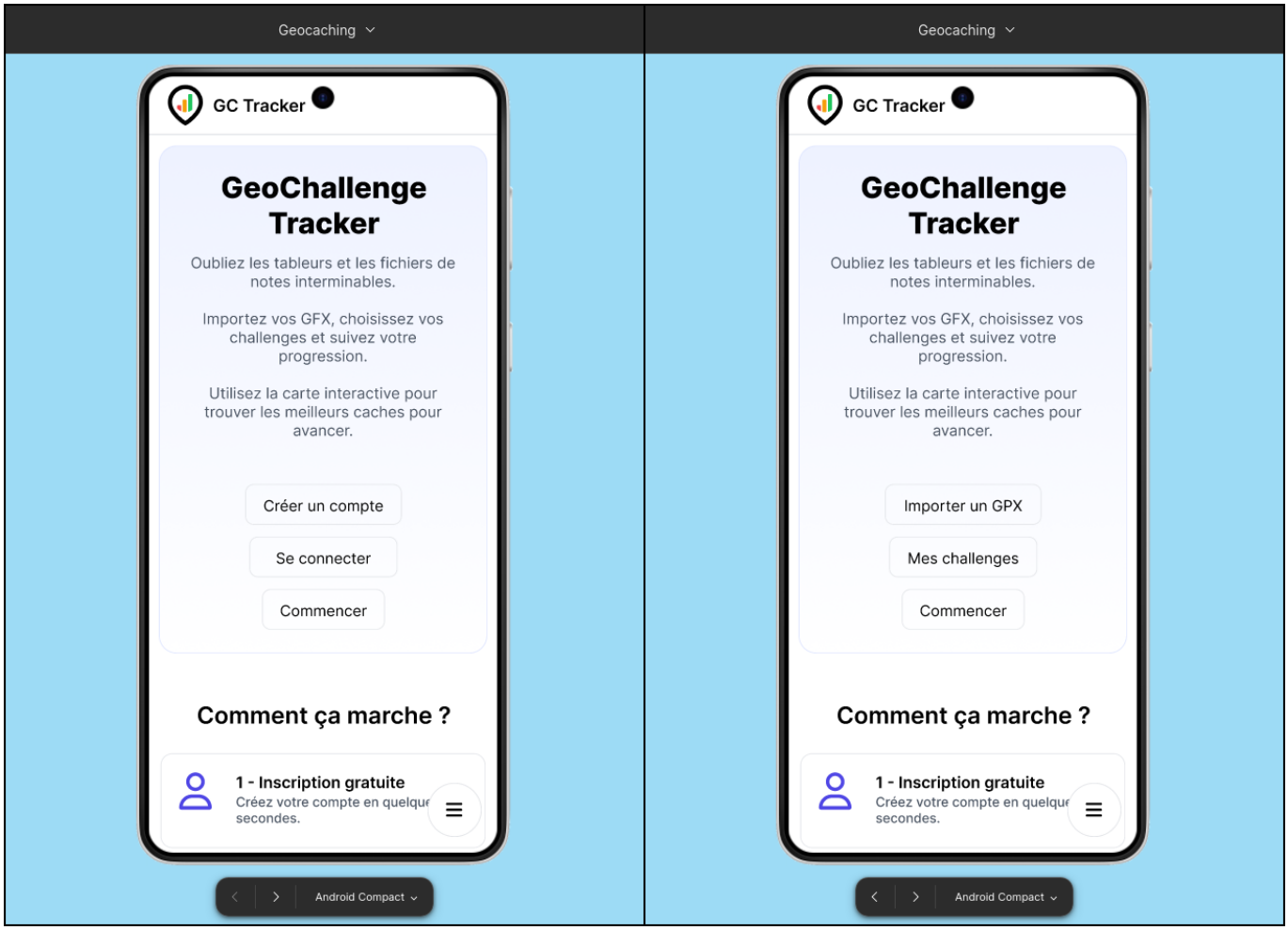


Figure 2 : Accueil non loggé / loggé

2.3 Inscription - Connexion

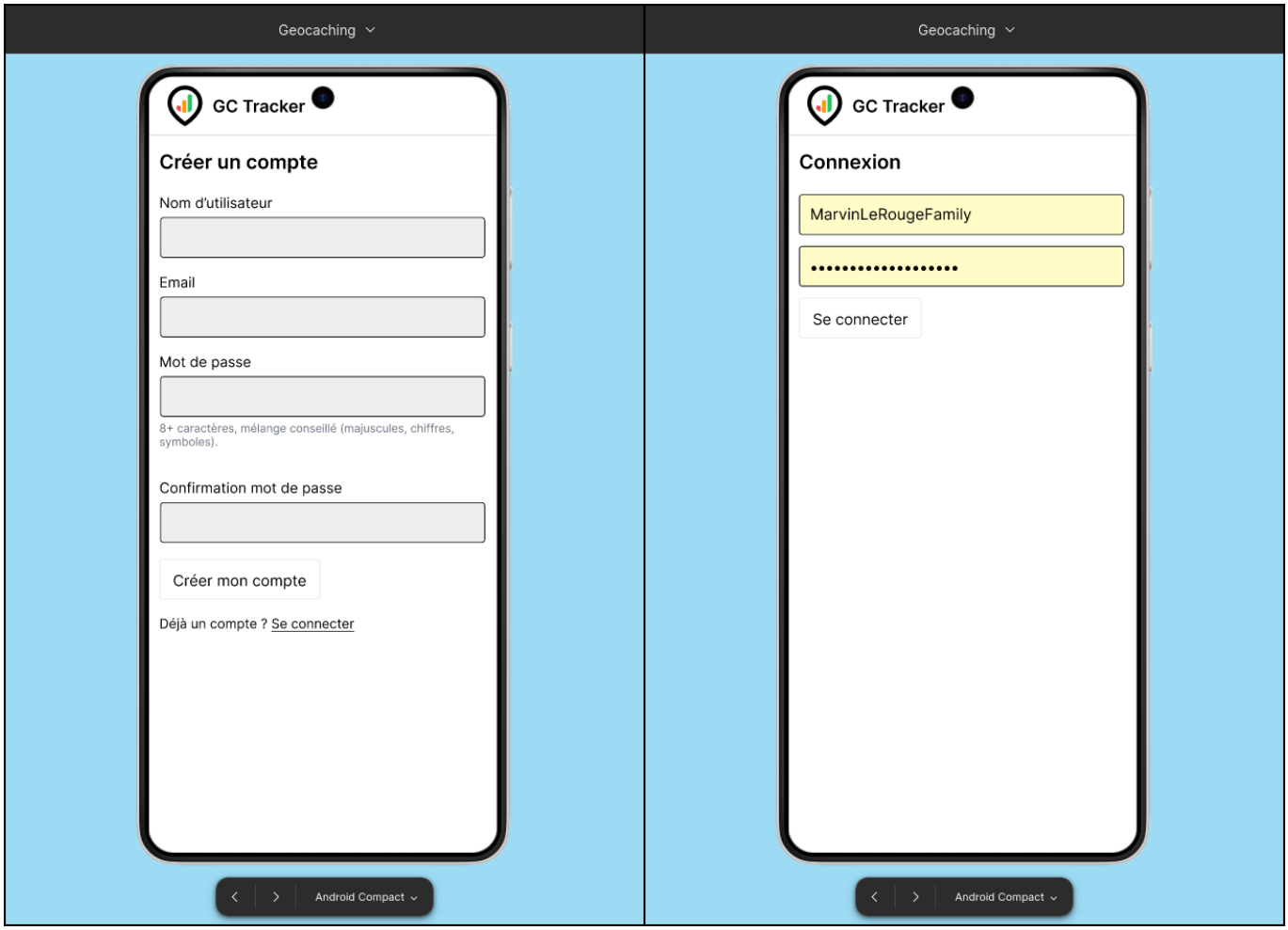


Figure 3 : Inscription / Connexion

3 Interfaces utilisateurs

3.1 Interactive map

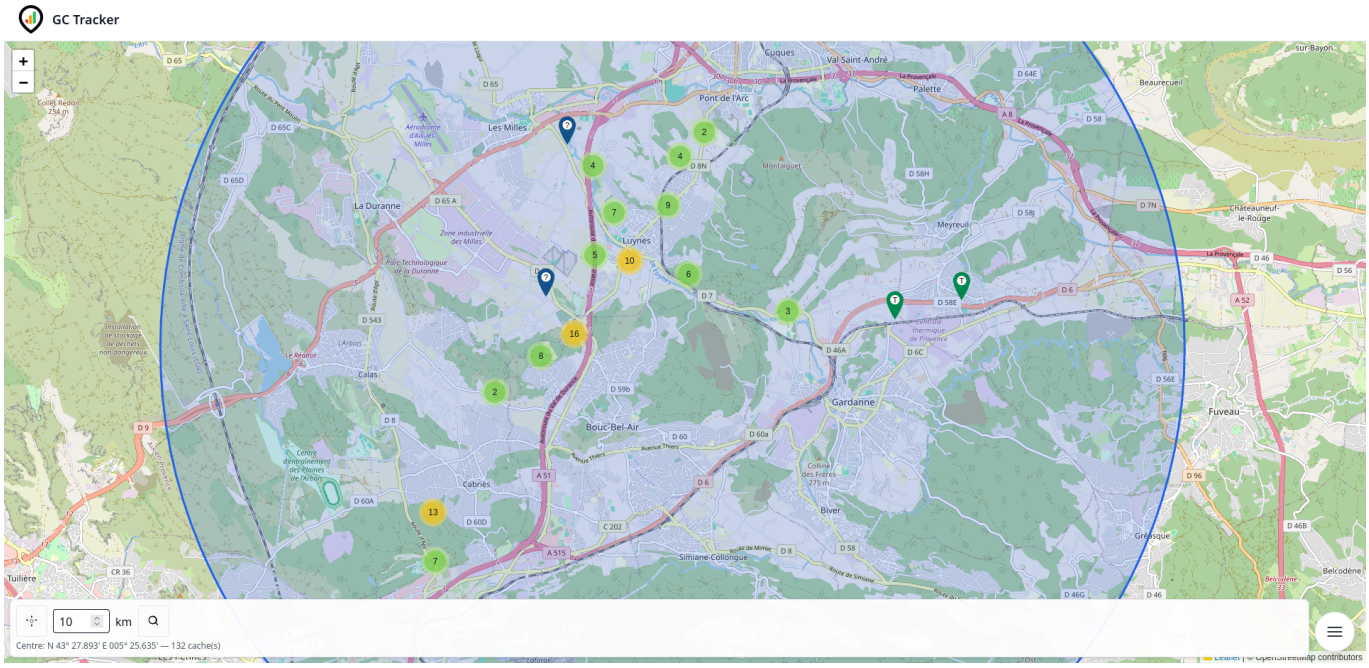


Figure 4 : Carte interactive, version desktop

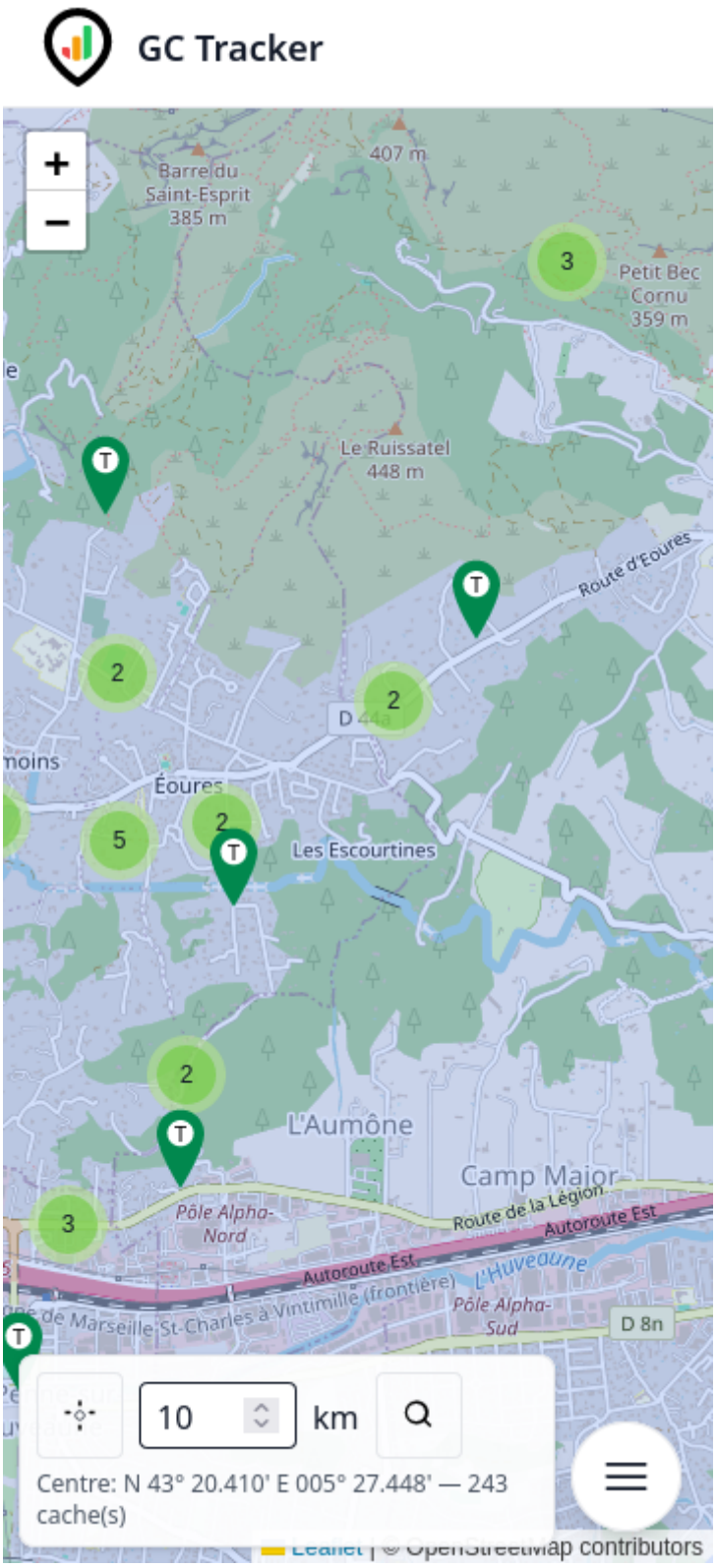


Figure 5 : Carte interactive, version mobile

3.2 Liste des challenges

Figure 6 : Liste des chalenges

GC Tracker

← Retour

18 - Rando Challenge - Le géologue régional

GC: GC9T6ZG

Créé: 02/10/2025 · Màj: 02/10/2025

Statut: **pending** (computed: completed) → effective: completed

Description

18 - Rando Challenge - Le géologue régional

Ce géoart , qui est composé de 41 caches challenge, vous fera faire une randonnée d'environ 10kms jusqu'à la calanque de l'Erevine. Les challenges qui composent cette série sont très variés en ce qui concerne les thèmes et les difficultés.
En ce qui concerne les boites, vous cherchez quasiment toujours le même style de contenant : des tube PET , à part pour certaines exceptions.
Il s'agit d'une cache challenge, la boite se trouve aux coordonnées indiquées dans le waypoint. Pas d'énigmes à résoudre mais des conditions à remplir.
Pour valider cette cache et la loguer found-it, vous devez avoir logué au moins 30 earthcaches dans le département des Bouches du Rhône. Voilà la liste des earthcaches des Bouches du Rhône : https://coord.info/BMACPRF

Pour savoir si vous êtes éligibles à un found-it, veuillez utiliser le checker ci-dessous.

Notes

Ajouter une note...

Raison d'override (optionnel)

Ex: contrôle manuel...

Enregistrer

Figure 7 : Challenge détails

4 Documentation API - GeoChallenge Tracker

4.1 Health

- GET `/ping` - Vérification de santé de l'API

4.2 Auth

- POST `/auth/register` - Inscription d'un nouvel utilisateur
- POST `/auth/login` - Connexion d'un utilisateur
- POST `/auth/refresh` - Renouvellement du token d'accès
- GET `/auth/verify-email` - Vérification d'email par code
- POST `/auth/verify-email` - Vérification d'email via POST
- POST `/auth/resend-verification` - Renvoi du code de vérification

4.3 Caches

- POST `/caches/upload-gpx` - Importe des caches depuis un fichier GPX/ZIP
- POST `/caches/by-filter` - Recherche de caches par filtres
- GET `/caches/within-bbox` - Caches dans une bounding box
- GET `/caches/within-radius` - Caches autour d'un point (rayon)
- GET `/caches/{gc}` - Récupère une cache par code GC
- GET `/caches/by-id/{id}` - Récupère une cache par identifiant MongoDB

4.4 Caches Elevation

- POST `/caches_elevation/caches/elevation/backfill` - Backfill de l'altitude manquante (admin)

4.5 Challenges

- POST `/challenges/refresh-from-caches` - (Re)crée les challenges depuis les caches 'challenge'

4.6 My Challenges

- POST `/my/challenges/sync` - Synchroniser les UserChallenges manquants
- GET `/my/challenges` - Lister mes UserChallenges
- PATCH `/my/challenges` - Patch en lot de plusieurs UserChallenges
- GET `/my/challenges/{uc_id}` - Détail d'un UserChallenge
- PATCH `/my/challenges/{uc_id}` - Modifier statut/notes d'un UserChallenge

4.7 My Challenge Tasks

- GET `/my/challenges/{uc_id}/tasks` - Lister les tâches d'un UserChallenge
- PUT `/my/challenges/{uc_id}/tasks` - Remplacer toutes les tâches d'un UserChallenge
- POST `/my/challenges/{uc_id}/tasks/validate` - Valider une liste de tâches (sans persistance)

4.8 My Challenge Progress

- GET `/my/challenges/{uc_id}/progress` - Obtenir le dernier snapshot et l'historique court
- POST `/my/challenges/{uc_id}/progress/evaluate` - Évaluer et enregistrer un snapshot immédiat
- POST `/my/challenges/new/progress` - Évaluer le premier snapshot pour les challenges sans progression

4.9 Targets

- POST `/my/challenges/{uc_id}/targets/evaluate` - Évaluer et persister les targets d'un UserChallenge
- GET `/my/challenges/{uc_id}/targets` - Lister les targets d'un UserChallenge
- DELETE `/my/challenges/{uc_id}/targets` - Supprimer toutes les targets d'un UserChallenge
- GET `/my/challenges/{uc_id}/targets/nearby` - Lister les targets proches d'un point (par UC)
- GET `/my/targets` - Lister toutes mes targets (tous challenges)
- GET `/my/targets/nearby` - Lister les targets proches d'un point (tous challenges)

4.10 My Profile

- GET `/my/profile/location` - Obtenir ma dernière localisation
- PUT `/my/profile/location` - Enregistrer ou mettre à jour ma localisation
- GET `/my/profile` - Obtenir mon profil

4.11 Maintenance

- GET `/maintenance` - Maintenance Get 1
- POST `/maintenance` - Maintenance Post 1

5 Composants métier

5.1 Parser GPX

```python
# backend/app/services/parsers/GPXCacheParser.py
# Parse un fichier GPX (ouvert depuis un chemin) pour extraire des géocaches structurées (
    ↪ métadonnées, attributs).

import html
from pathlib import Path
from typing import Any
from lxml import etree
from app.services.parsers.HTMLSanitizer import HTMLSanitizer

class GPXCacheParser:
    """Parseur GPX de géocaches.

    Description:
        Lit un fichier GPX (schémas `gpx`, `groundspeak`, `gsak`) et en extrait une
        liste de caches prêtes pour l'import : code GC, titre, coordonnées, type,
        taille, propriétaire, D/T, pays/état, description HTML (sanitisée), favoris,
        notes, dates (placement / found), attributs, etc.

    Attributes:
        gpx_file (Path): Chemin du fichier GPX.
        namespaces (dict): Préfixes d'espaces de noms XML utilisés pour les requêtes XPath.
        caches (list[dict]): Résultats accumulés après `parse()`.
        sanitizer (HTMLSanitizer): Sanitizeur HTML pour la description longue.
    """

    def __init__(self, gpx_file: Path):
        """Initialiser le parseur GPX.

        Description:
            Conserve le chemin vers le GPX, initialise les espaces de noms attendus
            et prépare les structures internes (liste `caches`, sanitizeur HTML).

        Args:
            gpx_file (Path): Chemin du fichier GPX à analyser.

        Returns:
            None
        """
        self.gpx_file = gpx_file
        self.namespaces = {
            "gpx": "http://www.topografix.com/GPX/1/0",
            "groundspeak": "http://www.groundspeak.com/cache/1/0/1",
            "gsak": "http://www.gsak.net/xmlv1/6",
        }
        self.caches: list[dict] = []
        self.sanitizer = HTMLSanitizer()

    def test(self):
        """Lister tous les tags XML (debug).

        Description:
            Parse le fichier et itère sur tous les éléments pour imprimer
            leurs `tag` (utilitaire de mise au point).

        Args:
            None

        Returns:
            None
        """
        tree = etree.parse(str(self.gpx_file))
        for elem in tree.getroot().iter():
            print(elem.tag)
```

```
    def parse(self) -> list[dict]:
        """Analyser le GPX et remplir `self.caches`.

        Description:
            - Parcourt les waypoints `//gpx:wpt` et cherche le sous-élément `groundspeak:cache`.\n
            - Pour chaque cache finale (`_is_final_waypoint`), extrait les champs utiles
              (GC, titre, coords, type, taille, owner, D/T, pays/état, description HTML
              nettoyée, favoris GSAK, notes, dates, attributs via `_parse_attributes`).\n
            - Empile chaque dict dans `self.caches`.

        Args:
            None

        Returns:
            list[dict]: Liste de caches structurées prêtes à l'import.
        """
        tree = etree.parse(str(self.gpx_file))
        nodes: Any = tree.xpath("//gpx:wpt", namespaces=self.namespaces)
        if not isinstance(nodes, list):
            raise ValueError("XPath did not return nodes")

        for wpt in nodes:
            cache_elem = wpt.find("groundspeak:cache", namespaces=self.namespaces)
            if cache_elem is None:
                continue

            is_final = self._is_final_waypoint(cache_elem)
            if is_final:
                cache = {
                    "GC": self.find_text_deep(wpt, "gpx:name"),
                    "title": self.find_text_deep(wpt, "gpx:desc"),
                    "latitude": float(wpt.attrib["lat"]),
                    "longitude": float(wpt.attrib["lon"]),
                    ...
                    "attributes": self._parse_attributes(cache_elem),
                }
                self.caches.append(cache)

        return self.caches

    def _parse_attributes(self, cache_elem) -> list[dict]:
        """Extraire la liste des attributs depuis `<groundspeak:attributes>`.

        Description:
            Parcourt les noeuds `groundspeak:attribute` et retourne des objets
            `{id: int, is_positive: bool, name: str}`.

        Args:
            cache_elem: Élément XML `<groundspeak:cache>` parent.

        Returns:
            list[dict]: Attributs normalisés (id / inc / libellé).
        """
        attrs = []
        for attr in cache_elem.xpath(
            "groundspeak:attributes/groundspeak:attribute", namespaces=self.namespaces
        ):
            attrs.append(
                {
                    "id": int(attr.get("id")),
                    "is_positive": attr.get("inc") == "1",
                    "name": attr.text.strip() if attr.text else "",
                }
            )

        return attrs

    def _has_corrected_coordinates(self, wpt_elem) -> bool:
```

```
        ...
    def _has_found_log(self, cache_elem) -> bool:
        ...
    def _was_found(self, wpt_elem) -> bool:
        ...
    def _is_final_waypoint(self, cache_elem) -> bool:
        ...
    def _text(self, element, default: str = "") -> str:
        ...
    def _html(self, element, default: str = "") -> str:
        ...
    def get_caches(self) -> list[dict]:
        ...
    def find_text_deep(self, element, tag: str) -> str:
        ...
```

5.2 Query builder

```
# backend/app/services/query_builder.py
# Transforme une expression canonique (AND-only) en conditions MongoDB pour la collection `caches
    ↪ `.

from __future__ import annotations
from datetime import date, datetime
from typing import Any
from bson import ObjectId
from app.services.referentials_cache import (resolve_attribute_code, resolve_country_name,
    ↪ resolve_size_code, resolve_size_name, esolve_state_name, resolve_type_code)

# NOTE: on ne dépend pas des modèles Pydantic ici : on reçoit un dict "expression" déjà canonisé
# (cf. services/user_challenge_tasks.put_tasks qui stocke l'expression canonicalisée). :
    ↪ contentReference[oaicite:1]{index=1}


def _mk_date(dt_or_str: Any) -> datetime:
    ...

def _flatten_and_nodes(expr: dict[str, Any]) -> list[dict[str, Any]] | None:
    """Aplatir récursivement les noeuds `AND` en une liste de feuilles.

    Description:
        Retourne `None` si l'expression contient des `OR`/`NOT` (non supportés par le compilateur
    ↪ « AND-only »).

    Args:
        expr (dict): Expression AST canonique.

    Returns:
        list[dict] | None: Feuilles si AND pur, sinon None.
    """
    kind = expr.get("kind")
    if kind == "and":
        out: list[dict[str, Any]] = []
        for n in expr.get("nodes") or []:
            sub = _flatten_and_nodes(n) if isinstance(n, dict) else [n]
            if sub is None:
                return None
            out.extend(sub)
        return out
    if kind in ("or", "not"):
        return None
    return [expr]  # leaf


def _extract_aggregate_spec(
    leaves: list[dict[str, Any]]
) -> tuple[dict[str, Any] | None, list[dict[str, Any]]]:
```

```
    """Extraire la spécification d'agrégat et les feuilles « cache.* ».

    Description:
        Détecte la **première** feuille d'agrégat parmi:
        - `aggregate_sum_difficulty_at_least`
        - `aggregate_sum_terrain_at_least`
        - `aggregate_sum_diff_plus_terr_at_least`
        - `aggregate_sum_altitude_at_least`
        Retourne `(agg_spec, leaves_sans_agrégat)`.

    Args:
        leaves (list[dict]): Feuilles AND.

    Returns:
        tuple[dict | None, list[dict]]: Spéc d'agrégat (ou None) et feuilles restantes.
    """
    agg = None
    cache_leaves: list[dict[str, Any]] = []
    for lf in leaves:
        k = lf.get("kind")
        if k in (
            "aggregate_sum_difficulty_at_least",
            "aggregate_sum_terrain_at_least",
            "aggregate_sum_diff_plus_terr_at_least",
            "aggregate_sum_altitude_at_least",
        ):
            if agg is None and lf.get("min_total") is not None:
                mt = int(lf["min_total"])
                if k == "aggregate_sum_difficulty_at_least":
                    agg = {"kind": "difficulty", "min_total": mt}
                elif k == ...
        else:
            cache_leaves.append(lf)
    return agg, cache_leaves


def _compile_leaf_to_cache_pairs(leaf: dict[str, Any]) -> list[tuple[str, Any]]:
    """Compiler une feuille AST en `(champ, condition)` sur `caches`.

    Description:
        Supporte notamment:
        - `type_in`, `size_in` (résolution via référentiels/aliases)
        - `country_is`, `state_in`
        - `placed_year`, `placed_before`, `placed_after`
        - `difficulty_between`, `terrain_between`
        - `attributes` (±, `attributes.$elemMatch`)

    Args:
        leaf (dict): Feuille individuelle.

    Returns:
        list[tuple[str, Any]]: Paires `(champ, condition)` à fusionner en AND.
    """
    k = leaf.get("kind")
    out: list[tuple[str, Any]] = []

    oids: list[ObjectId] = []
    if k == "type_in":
        # 1) canonique: types: [{cache_type_doc_id | cache_type_id | cache_type_code}]
        for t in leaf.get("types") or []:
            oid = t.get("cache_type_doc_id")
            if not oid and t.get("cache_type_id") is not None:
                # numeric id non supporté nativement par le cache -> on ignore, ou ajoute si tu l'
    ↪ as dans cache
                pass
            if not oid and t.get("cache_type_code"):
                oid = resolve_type_code(t["cache_type_code"])
            if oid:
                oids.append(oid)
```

```
        if oids:
            out.append(("type_id", {"$in": list(dict.fromkeys(oids))}))
        return out

    # Traitement des cas size_in / country_is / state_in / placed_year / placed_before /
    ↪ placed_after / difficulty_between / terrain_between

    if k == "attributes":
        # Canonique: [{"cache_attribute_doc_id"| "cache_attribute_id" | "code", "is_positive":
    ↪ bool}]
        attrs = leaf.get("attributes") or []
        for a in attrs:
            is_pos = bool(a.get("is_positive", True))
            attr_oid = a.get("cache_attribute_doc_id") or a.get("attribute_doc_id")
            if not attr_oid and a.get("cache_attribute_id") is not None:
                # le cache retourne aussi l'id numérique via resolve_attribute_code(code) si tu
    ↪ veux;
                # ici on reste doc_id only
                pass
            if not attr_oid and a.get("code"):
                res = resolve_attribute_code(a["code"])
                attr_oid = res[0] if res else None

            if attr_oid:
                out.append(
                    (
                        "attributes",
                        {
                            "$elemMatch": {
                                "attribute_doc_id": ObjectId(str(attr_oid)),
                                "is_positive": is_pos,
                            }
                        },
                    )
                )
            else:
                out.append(("_id", ObjectId()))  # clause impossible

        return out

    return out


def compile_and_only(
    expr: dict[str, Any]
) -> tuple[str, dict[str, Any], bool, list[str], dict[str, Any] | None]:
    """Compiler une expression AND en filtres Mongo « caches.* ».

    Description:
        - Rejette `OR`/`NOT` (`supported=False`, notes).\n
        - Extrait un éventuel agrégat (diff/terr/diff+terr/altitude).\n
        - Compile chaque feuille en paires `(champ, condition)` et fusionne par champ (AND).\n
        - Génère une signature stable de l'expression (`"and:" + json.dumps(leaves)`).

    Args:
        expr (dict): Expression canonique.

    Returns:
        tuple:
            str: Signature compilée.
            dict: `match_caches` — conditions AND par champ.
            bool: `supported` — True si AND pur.
            list[str]: `notes` — avertissements/causes de non-support.
            dict | None: `aggregate_spec` — spécification d'agrégat.
    """
    leaves = _flatten_and_nodes(expr)
    if leaves is None:
        return ("unsupported:or-not", {}, False, ["or/not unsupported in MVP"], None)
```

```
    agg_spec, cache_leaves = _extract_aggregate_spec(leaves)
    parts: list[tuple[str, Any]] = []
    for lf in cache_leaves:
        parts.extend(_compile_leaf_to_cache_pairs(lf))

    # fusion (AND): grouper par champ; si plusieurs conds pour un même champ -> liste ET-ée
    match: dict[str, Any] = {}
    for field, cond in parts:
        if field in match:
            if not isinstance(match[field], list):
                match[field] = [match[field]]
            match[field].append(cond)
        else:
            match[field] = cond

    try:
        import json

        signature = "and:" + json.dumps({"leaves": cache_leaves}, default=str, sort_keys=True)
    except Exception:
        signature = "and:compiled"

    return (signature, match, True, [], agg_spec)
```

5.3 Calcul de snapshot progress

```
# backend/app/services/progress.py
# Calcule des snapshots de progression par UserChallenge, mise à jour des statuts, et accès à l'
    ↪ historique.

from __future__ import annotations
import math
from datetime import date, datetime, timedelta
from typing import Any
from bson import ObjectId
from pymongo import ASCENDING, DESCENDING
from app.core.utils import now, utcnow
from app.db.mongodb import get_collection
from app.services.query_builder import compile_and_only

# ---------- Helpers ----------


def _ensure_uc_owned(user_id: ObjectId, uc_id: ObjectId) -> dict[str, Any]:
    """Vérifier que l'UC appartient bien à l'utilisateur.

    Description:
        Contrôle l'existence de `user_challenges[_id=uc_id, user_id=user_id]`. Lève en cas de non-
    ↪ appartenance.

    Args:
        user_id (ObjectId): Identifiant utilisateur.
        uc_id (ObjectId): Identifiant UserChallenge.

    Returns:
        dict: Document minimal (_id) si autorisé.

    Raises:
        PermissionError: Si l'UC n'appartient pas à l'utilisateur (ou n'existe pas).
    """
    ucs = get_collection("user_challenges")
    row = ucs.find_one({"_id": uc_id, "user_id": user_id}, {"_id": 1})
    if not row:
        raise PermissionError("UserChallenge not found or not owned by user")
    return row
```

```
def _get_tasks_for_uc(uc_id: ObjectId) -> list[dict[str, Any]]:
    ...
def _attr_id_by_cache_attr_id(cache_attribute_id: int) -> ObjectId | None:
    ...

def _count_found_caches_matching(user_id: ObjectId, match_caches: dict[str, Any]) -> int:
    """Compter les trouvailles d'un utilisateur qui matchent des conditions « caches.* ».

    Description:
        Pipeline: filtre par `user_id` sur `found_caches`, `$lookup` vers `caches`, `$unwind`,
        puis application des conditions (`match_caches`) sur `cache.*`, et `$count`.

    Args:
        user_id (ObjectId): Utilisateur concerné.
        match_caches (dict): Conditions AND sur des champs de `caches`.

    Returns:
        int: Nombre de trouvailles correspondantes.
    """
    fc = get_collection("found_caches")
    pipeline: list[dict[str, Any]] = [
        {"$match": {"user_id": user_id}},
        {
            "$lookup": {
                "from": "caches",
                "localField": "cache_id",
                "foreignField": "_id",
                "as": "cache",
            }
        },
        {"$unwind": "$cache"},
    ]

    # Apply match on cache.*
    conds: list[dict[str, Any]] = []
    for field, cond in match_caches.items():
        if isinstance(cond, list):
            # multiple conditions for the same field => all must hold
            for c in cond:
                conds.append({f"cache.{field}": c})
        else:
            conds.append({f"cache.{field}": cond})
    if conds:
        pipeline.append({"$match": {"$and": conds}})
    pipeline.append({"$count": "current_count"})
    rows = list(fc.aggregate(pipeline, allowDiskUse=False))
    return int(rows[0]["current_count"]) if rows else 0


def _aggregate_total(user_id: ObjectId, match_caches: dict[str, Any], spec: dict[str, Any]) -> int
    ↪ :
    """Calculer une somme agrégée (difficulté, terrain, diff+terr, altitude).

    Description:
        Filtre via `match_caches` puis somme la métrique demandée :
        - `difficulty` → somme des difficultés
        - `terrain` → somme des terrains
        - `diff_plus_terr` → somme (difficulté + terrain)
        - `altitude` → somme des altitudes

    Args:
        user_id (ObjectId): Utilisateur.
        match_caches (dict): Conditions AND sur `caches`.
        spec (dict): Spécification d'agrégat (`{'kind': ..., 'min_total': int}`).

    Returns:
        int: Total agrégé (0 si `kind` inconnu).
    """
```

```
    fc = get_collection("found_caches")
    pipeline: list[dict[str, Any]] = [
        {"$match": {"user_id": user_id}},
        {
            "$lookup": {
                "from": "caches",
                "localField": "cache_id",
                "foreignField": "_id",
                "as": "cache",
            }
        },
        {"$unwind": "$cache"},
    ]
    # Apply match on cache.*
    conds: list[dict[str, Any]] = []
    for field, cond in match_caches.items():
        if isinstance(cond, list):
            for c in cond:
                conds.append({f"cache.{field}": c})
        else:
            conds.append({f"cache.{field}": cond})
    if conds:
        pipeline.append({"$match": {"$and": conds}})

    k = spec["kind"]
    if k == "difficulty":
        score_expr = {"$ifNull": ["$cache.difficulty", 0]}
    elif k == "terrain":
        score_expr = {"$ifNull": ["$cache.terrain", 0]}
    elif k == "diff_plus_terr":
        score_expr = {
            "$add": [
                {"$ifNull": ["$cache.difficulty", 0]},
                {"$ifNull": ["$cache.terrain", 0]},
            ]
        }
    elif k == "altitude":
        score_expr = {"$ifNull": ["$cache.elevation", 0]}
    else:
        return 0

    pipeline += [
        {"$project": {"score": score_expr}},
        {"$group": {"_id": None, "total": {"$sum": "$score"}}},
    ]
    rows = list(fc.aggregate(pipeline, allowDiskUse=False))
    return int(rows[0]["total"]) if rows else 0


def _nth_found_date(user_id: ObjectId, match_caches: dict[str, Any], n: int) -> date | None:
    ...

def evaluate_progress(user_id: ObjectId, uc_id: ObjectId, force=False) -> dict[str, Any]:
    """Évaluer les tâches d'un UC et insérer un snapshot.

    Description:
        - Vérifie l'appartenance de l'UC (`_ensure_uc_owned`).\n
        - Si `force=False` et que l'UC est déjà `completed`, retourne le dernier snapshot (si
    ↪ existant).\n
        - Pour chaque tâche, compile l'expression (`compile_and_only`), compte les trouvailles,
    ↪ met à jour
          éventuellement le statut de la tâche, calcule les agrégats et le pourcentage.\n
        - Calcule l'agrégat global et crée un document `progress`. Si toutes les tâches supportées
    ↪  sont `done`,
          met à jour `user_challenges` en `completed` (statuts déclaré & calculé).

    Args:
        user_id (ObjectId): Utilisateur.
        uc_id (ObjectId): UserChallenge.
```

```
        force (bool): Forcer le recalcul même si UC complété.

    Returns:
        dict: Document snapshot inséré (avec `id` ajouté pour la réponse).
    """
    _ensure_uc_owned(user_id, uc_id)
    tasks = _get_tasks_for_uc(uc_id)
    snapshots: list[dict[str, Any]] = []
    sum_current = 0
    sum_min = 0
    tasks_supported = 0
    tasks_done = 0
    uc_statuses = get_collection("user_challenges").find_one(
        {"_id": uc_id}, {"status": 1, "computed_status": 1}
    )
    uc_status = (uc_statuses or {}).get("status")
    uc_computed_status = (uc_statuses or {}).get("computed_status")
    if (not force) and (uc_computed_status == "completed" or uc_status == "completed"):
        # Renvoyer le dernier snapshot existant, sans recalcul ni insertion
        last = get_collection("progress").find_one(
            {"user_challenge_id": uc_id}, sort=[("checked_at", -1), ("created_at", -1)]
        )
        if last:
            return last  # même shape que vos snapshots persistés
        # S'il n'y a pas encore de snapshot, on retombe sur le calcul normal

    for t in tasks:
        min_count = int((t.get("constraints") or {}).get("min_count") or 0)
        title = t.get("title") or "Task"
        order = int(t.get("order") or 0)
        status = (t.get("status") or "todo").lower()
        expr = t.get("expression") or {}

        if status == "done" and not force:
            snap = {
                "task_id": t["_id"],
                "order": order,
                "title": title,
                "status": status,
                "supported_for_progress": True,
                "compiled_signature": "override:done",
                "min_count": min_count,
                "current_count": min_count,
                "percent": 100.0,
                "notes": ["user override: done"],
                "evaluated_in_ms": 0,
                "last_evaluated_at": now(),
                "updated_at": t.get("updated_at"),
                "created_at": t.get("created_at"),
            }
        else:
            sig, match_caches, supported, notes, agg_spec = compile_and_only(expr)
            if not supported:
                snap = {
                    "task_id": t["_id"],
                    "order": order,
                    "title": title,
                    "supported_for_progress": False,
                    ...
                }
            else:
                tic = utcnow()
                current = _count_found_caches_matching(user_id, match_caches)
                ms = int((utcnow() - tic).total_seconds() * 1000)

                # base percent on min_count
                bounded = min(current, min_count) if min_count > 0 else current
                count_percent = (100.0 * (bounded / min_count)) if min_count > 0 else 100.0
                new_status = "done" if current >= min_count else status
```

```
            task_id = t["_id"]
            t["status"] = new_status
            if status != "done":
                get_collection("user_challenge_tasks").update_one(
                    {"_id": task_id},
                    {
                        "$set": {
                            "status": new_status,
                            "last_evaluated_at": utcnow(),
                            "updated_at": utcnow(),
                        }
                    },
                )

            # aggregate handling
            aggregate_total = None
            aggregate_target = None
            aggregate_percent = None
            aggregate_unit = None
            if agg_spec:
                aggregate_total = _aggregate_total(user_id, match_caches, agg_spec)
                aggregate_target = int(agg_spec.get("min_total", 0)) or None
                if aggregate_target and aggregate_target > 0:
                    aggregate_percent = max(
                        0.0,
                        min(
                            100.0,
                            100.0 * (float(aggregate_total) / float(aggregate_target)),
                        ),
                    )
                else:
                    aggregate_percent = None
                # unit: altitude -> meters, otherwise points
                aggregate_unit = "meters" if agg_spec.get("kind") == "altitude" else "points"

            # final percent rule (MVP):
            # - if both count & aggregate constraints exist -> percent = min(count_percent,
↪ aggregate_percent)
            # - if only count -> count_percent
            # - if only aggregate -> aggregate_percent or 0 if None
            if agg_spec and min_count > 0:
                final_percent = min(count_percent, (aggregate_percent or 0.0))
            elif agg_spec and min_count == 0:
                final_percent = aggregate_percent or 0.0
            else:
                final_percent = count_percent

            # --- dates de progression persistées sur la task ---
            task_id = t["_id"]
            min_count = int((t.get("constraints") or {}).get("min_count") or 0)

            # 2.1 start_found_at : première trouvaille qui matche
            start_dt = _first_found_date(user_id, match_caches)
            if start_dt and not t.get("start_found_at"):
                get_collection("user_challenge_tasks").update_one(
                    {"_id": task_id},
                    {"$set": {"start_found_at": start_dt, "updated_at": utcnow()}},
                )
                t["start_found_at"] = start_dt  # en mémoire pour la suite

            # 2.2 completed_at : date de la min_count-ième trouvaille
            completed_dt = None
            if min_count > 0 and current >= min_count:
                completed_dt = _nth_found_date(user_id, match_caches, min_count)

            # persister la date si atteinte, sinon l'annuler si elle existait mais plus valide
            if completed_dt:
                if t.get("completed_at") != completed_dt:
                    get_collection("user_challenge_tasks").update_one(
```

```
                            {"_id": task_id},
                            {
                                "$set": {
                                    "completed_at": completed_dt,
                                    "updated_at": utcnow(),
                                }
                            },
                        )
                        t["completed_at"] = completed_dt
                else:
                    if t.get("completed_at") is not None:
                        get_collection("user_challenge_tasks").update_one(
                            {"_id": task_id},
                            {"$set": {"completed_at": None, "updated_at": utcnow()}},
                        )
                        t["completed_at"] = None

                snap = {
                    "task_id": t["_id"],
                    "order": order,
                    "title": title,
                    "status": t["status"],
                    "supported_for_progress": True,
                    "compiled_signature": sig,
                    "min_count": min_count,
                    "current_count": current,
                    "percent": final_percent,
                    # per-task aggregate block for DTO:
                    "aggregate": (
                        None
                        if not agg_spec
                        else {
                            "total": aggregate_total,
                            "target": aggregate_target or 0,
                            "unit": aggregate_unit or "points",
                        }
                    ),
                    "notes": notes,
                    "evaluated_in_ms": ms,
                    "last_evaluated_at": now(),
                    "updated_at": t.get("updated_at"),
                    "created_at": t.get("created_at"),
                }

            if snap["supported_for_progress"]:
                tasks_supported += 1
                sum_min += max(0, min_count)
                bounded_for_sum = (
                    min(snap["current_count"], min_count) if min_count > 0 else snap["current_count"]
                )
                sum_current += bounded_for_sum
                if bounded_for_sum >= min_count and min_count > 0:
                    tasks_done += 1

            snapshots.append(snap)

        aggregate_percent = (100.0 * (sum_current / sum_min)) if sum_min > 0 else 0.0
        aggregate_percent = round(aggregate_percent, 1)
        doc = {
            "user_challenge_id": uc_id,
            "checked_at": now(),
            "aggregate": {
                "percent": aggregate_percent,
                "tasks_done": tasks_done,
                "tasks_total": tasks_supported,
                "checked_at": now(),
            },
            "tasks": snapshots,
            "message": None,
```

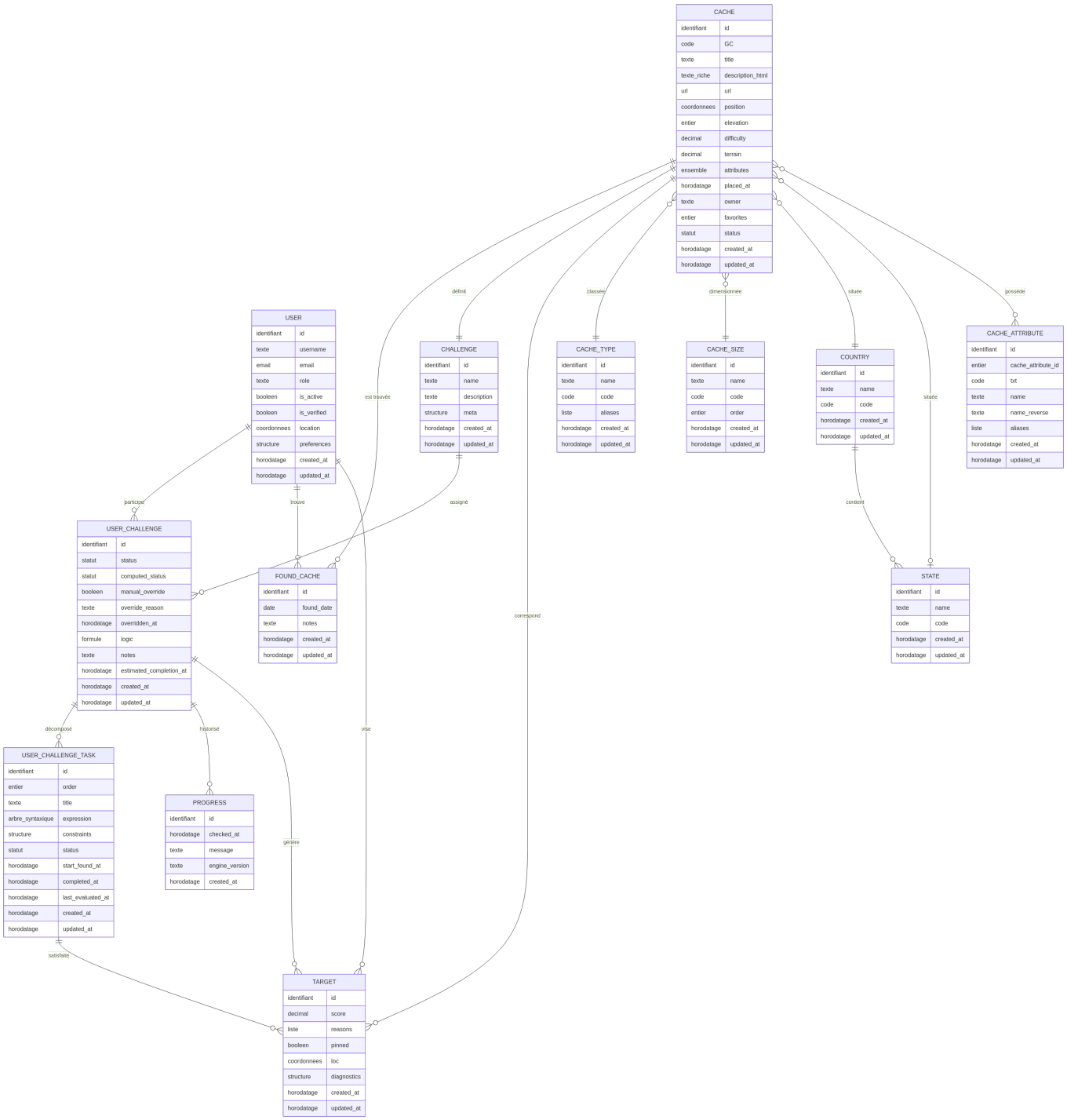
```
        "created_at": now(),
    }
    if (uc_computed_status != "completed") and (tasks_done == tasks_supported):
        new_status = "completed"
        get_collection("user_challenges").update_one(
            {"_id": uc_id},
            {
                "$set": {
                    "computed_status": new_status,
                    "status": new_status,
                    "updated_at": utcnow(),
                }
            },
        )
    get_collection("progress").insert_one(doc)
    # enrich for response
    doc["id"] = str(doc.get("_id")) if "_id" in doc else None

    return doc


def get_latest_and_history(
    user_id: ObjectId,
    uc_id: ObjectId,
    limit: int = 10,
    before: datetime | None = None,
) -> dict[str, Any]:
    """Obtenir le dernier snapshot et un historique court.

    Description:
        Récupère jusqu'à `limit` snapshots (tri desc), renvoie le plus récent et un historique
        résumé (date + agrégat). `before` permet de paginer en arrière.

    Args:
        user_id (ObjectId): Utilisateur.
        uc_id (ObjectId): UserChallenge.
        limit (int): Taille max de l'historique ≥(1).
        before (datetime | None): Curseur temporel exclusif.

    Returns:
        dict: `{'latest': dict | None, 'history': list[dict]}`.
    """
    q: dict[str, Any] = {}
    _ensure_uc_owned(user_id, uc_id)
    coll = get_collection("progress")
    q = {"user_challenge_id": uc_id}
    if before:
        q["checked_at"] = {"$lt": before}
    cur = coll.find(q).sort([("checked_at", DESCENDING)]).limit(limit)
    items = list(cur)
    latest = items[0] if items else None
    history = items[1:] if len(items) > 1 else []

    # --- enrichir 'latest' avec ETA par tâche + ETA globale ---
    if latest:
        # map (task_id -> {start_found_at, completed_at, min_count courant})
        tasks_coll = get_collection("user_challenge_tasks")
        tdocs = list(
            tasks_coll.find(
                {"user_challenge_id": uc_id},
                {"_id": 1, "start_found_at": 1, "completed_at": 1, "constraints": 1},
            )
        )
        dates_by_tid: dict[ObjectId, dict[str, Any]] = {
            d["_id"]: {
                "start": d.get("start_found_at"),
                "done": d.get("completed_at"),
                "min_count": int((d.get("constraints") or {}).get("min_count") or 0),
            }
```

```
            for d in tdocs
        }

        # calcule ETA par tâche du snapshot 'latest' en fonction d'aujourd'hui
        now_dt = now()
        eta_values: list[datetime] = []
        for it in latest.get("tasks") or []:
            tid = it.get("task_id")
            cur = int(it.get("current_count") or 0)
            # min_count : priorité au snapshot si présent, sinon doc task
            min_c = int(it.get("min_count") or dates_by_tid.get(tid, {}).get("min_count") or 0)
            info = dates_by_tid.get(tid) or {}
            start = info.get("start")
            done = info.get("done")

            eta = None
            if done:
                # terminé -> ETA figée
                # found_date est un 'date', on le normalise en 'datetime' pour la réponse
                eta = datetime(done.year, done.month, done.day)  # 00:00 locale/UTC selon now()
            elif start and cur >= 1 and min_c > 0:
                # progression -> extrapolation
                # vitesse = (cur - 1) / jours écoulés depuis la 1ère trouvaille
                elapsed_days = max((now_dt.date() - start.date()).days, 1)
                speed = float(cur - 1) / float(elapsed_days)
                remaining = max(0, min_c - cur)
                if speed > 0.0 and remaining > 0:
                    eta_days = int(math.ceil(remaining / speed))
                    eta_date = now_dt.date() + timedelta(days=eta_days)
                    eta = datetime(eta_date.year, eta_date.month, eta_date.day)
                # sinon, eta = None

            # injecter l'ETA par tâche dans l'objet 'latest' (pour DTO)
            it["estimated_completion_at"] = eta

            if eta:
                eta_values.append(eta)

        # ETA globale = max des ETA non-None
        latest.setdefault("aggregate", {})
        latest["estimated_completion_at"] = max(eta_values) if eta_values else None

    def _summarize(d: dict[str, Any]) -> dict[str, Any]:
        return {
            "checked_at": d["checked_at"],
            "aggregate": d["aggregate"],
        }

    res = {
        "latest": latest,
        "history": [_summarize(h) for h in history],
    }
    if latest and "_id" in latest:
        latest["id"] = str(latest["_id"])
    return res


def evaluate_new_progress(
    user_id: ObjectId,
    *,
    include_pending: bool = False,
    limit: int = 50,
    since: datetime | None = None,
) -> dict[str, Any]:
    """Évaluer un premier snapshot pour les UC sans progression.

    Description:
        Sélectionne les UC de l'utilisateur avec statut `accepted` (et `pending` si demandé),
        optionnellement créés depuis `since`, **ignore** ceux ayant déjà du `progress`,
```

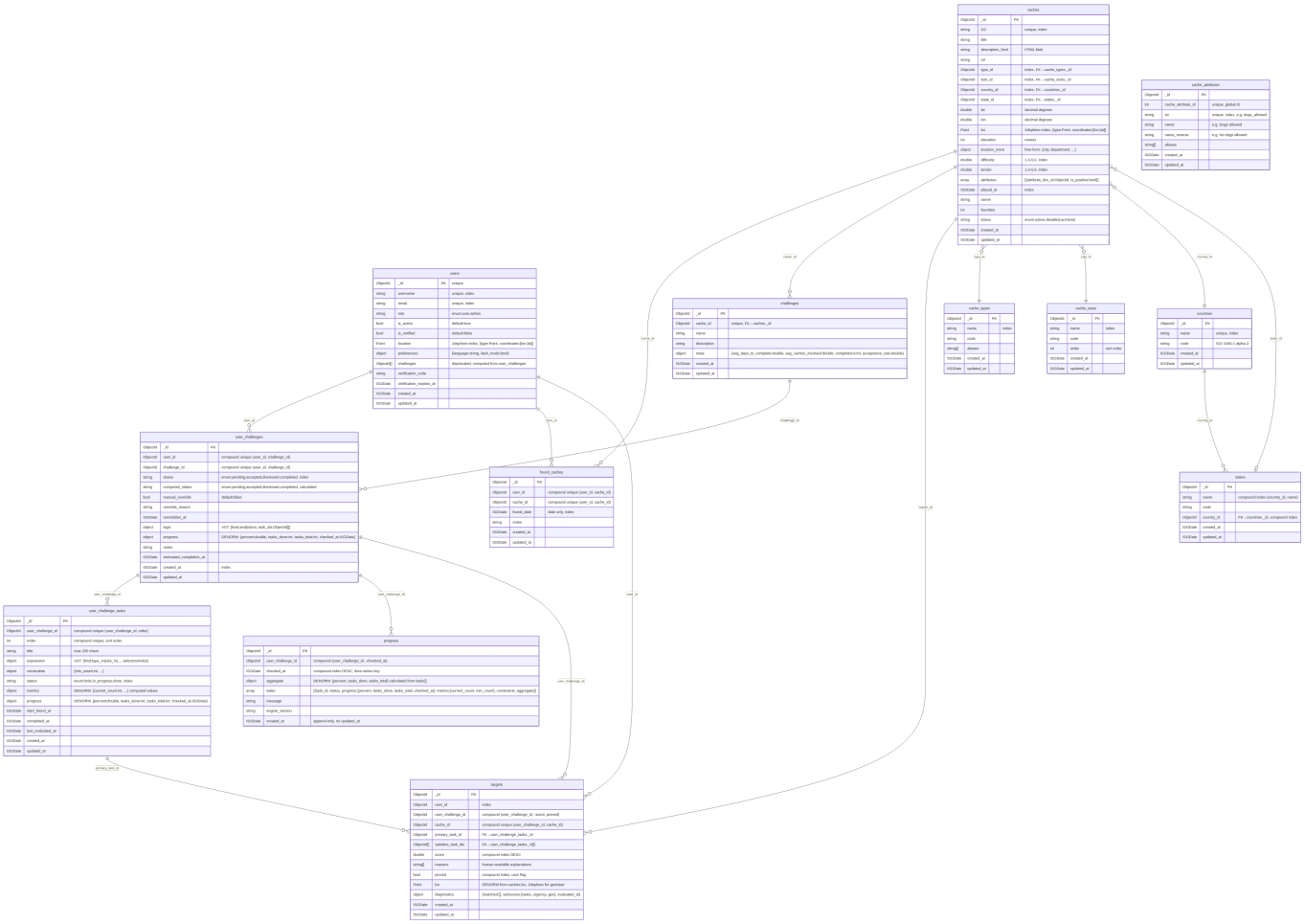
```
        puis évalue jusqu'à `limit` items.

    Args:
        user_id (ObjectId): Utilisateur.
        include_pending (bool): Inclure les UC `pending`.
        limit (int): Nombre max d'UC à traiter.
        since (datetime | None): Filtre de date de création.

    Returns:
        dict: `{'evaluated_count': int, 'skipped_count': int, 'uc_ids': list[str]}`.
    """
    ucs = get_collection("user_challenges")
    progress = get_collection("progress")

    st = ["accepted"] + (["pending"] if include_pending else [])
    q: dict[str, Any] = {"user_id": user_id, "status": {"$in": st}}
    if since:
        q["created_at"] = {"$gte": since}

    # candidates
    cand = list(ucs.find(q, {"_id": 1}).sort([("_id", ASCENDING)]).limit(limit * 3))
    uc_ids = [c["_id"] for c in cand]

    # remove those already in progress
    if not uc_ids:
        return {"evaluated_count": 0, "skipped_count": 0, "uc_ids": []}
    present = set(
        d["user_challenge_id"]
        for d in progress.find({"user_challenge_id": {"$in": uc_ids}}, {"user_challenge_id": 1})
    )
    todo = [uc_id for uc_id in uc_ids if uc_id not in present][:limit]

    evaluated_ids: list[str] = []
    for uc_id in todo:
        evaluate_progress(user_id, uc_id)
        evaluated_ids.append(str(uc_id))

    return {
        "evaluated_count": len(evaluated_ids),
        "skipped_count": len(uc_ids) - len(evaluated_ids),
        "uc_ids": evaluated_ids,
    }
```

6 Composants d’accès aux données

6.1 Modèle conceptuel de données



6.2 Modèle physique de données



6.3 Modèle Cache

```python
# backend/app/models/cache.py
# Modèle principal d'une géocache (métadonnées, typage, localisation, attributs, stats).

from __future__ import annotations
import datetime as dt
from typing import Any, Literal
from pydantic import BaseModel, ConfigDict, Field
from app.core.bson_utils import MongoBaseModel, PyObjectId
from app.core.utils import now

class CacheAttributeRef(BaseModel):
    """Référence d'attribut de cache.

    Description:
        Lien vers un document `cache_attributes` avec indication du sens (positif/négatif).

    Attributes:
        attribute_doc_id (PyObjectId): Référence à `cache_attributes._id`.
        is_positive (bool): True si l'attribut est affirmatif, False s'il est négatif.
    """

    attribute_doc_id: PyObjectId  # référence à cache_attributes._id
    is_positive: bool  # attribut positif (True) ou négatif (False)

    # Sous-modèle: ajouter model_config pour gérer PyObjectId partout (nested)
    model_config = ConfigDict(arbitrary_types_allowed=True, json_encoders={PyObjectId: str})

class CacheBase(BaseModel):
    """Champs de base d'une géocache.

    Description:
        Structure commune pour la création/lecture des caches : identifiants GC, typage,
        localisation (lat/lon + GeoJSON), attributs, difficultés/terrain, dates et stats.
    """

    GC: str
    title: str
    description_html: str | None = None
    url: str | None = None
    # Typage / classement
    type_id: PyObjectId | None = None  # ref -> CacheType
    size_id: PyObjectId | None = None  # ref -> CacheSize
    # Localisation
    country_id: PyObjectId | None = None  # ref -> Country
    state_id: PyObjectId | None = None  # ref -> State
    lat: float | None = None
    lon: float | None = None
    # GeoJSON pour index 2dsphere (coordonnées [lon, lat])
    loc: dict[str, Any] | None = None
    elevation: int | None = None  # en mètres (optionnel)
    location_more: dict[str, Any] | None = None  # infos libres (ville, département...)
    # Caractéristiques
    difficulty: float | None = None  # 1.0 .. 5.0
    terrain: float | None = None  # 1.0 .. 5.0
    attributes: list[CacheAttributeRef] = Field(default_factory=list)
    # Dates & stats
    placed_at: dt.datetime | None = None
    owner: str | None = None
    favorites: int | None = None
    status: Literal["active", "disabled", "archived"] | None = None

# class CacheCreate / CacheUpdate

class Cache(MongoBaseModel, CacheBase):
    """Document Mongo d'une géocache (avec horodatage).
```

```
    Description:
        Étend `CacheBase` avec les champs de traçabilité (_id, created_at, updated_at).
    """

    created_at: dt.datetime = Field(default_factory=lambda: now())
    updated_at: dt.datetime | None = None
```

6.4 Modèle User Challenge

```
# backend/app/models/user_challenge.py
# État d'un challenge pour un utilisateur (statuts déclarés/calculés, logique UC, notes, progress)
    ↪ .

from __future__ import annotations
import datetime as dt
from typing import Literal
from pydantic import Field
from app.core.bson_utils import MongoBaseModel, PyObjectId
from app.core.utils import now
from app.models._shared import ProgressSnapshot
from app.models.challenge_ast import UCLogic

class UserChallenge(MongoBaseModel):
    """Document Mongo « UserChallenge ».

    Description:
        Lie un utilisateur à un challenge, stocke le statut utilisateur (déclaratif) et le
        statut calculé (évaluation UC logic), ainsi que l'override manuel et un snapshot courant.

    Attributes:
        user_id (PyObjectId): Réf. utilisateur.
        challenge_id (PyObjectId): Réf. challenge.
        status (Literal['pending','accepted','dismissed','completed']): Statut déclaré.
        computed_status (Literal[...] | None): Statut calculé.
        manual_override (bool): Override manuel actif.
        override_reason (str | None): Justification d'override.
        overridden_at (datetime | None): Date override.
        logic (UCLogic | None): Logique d'agrégation des tasks.
        progress (ProgressSnapshot | None): Snapshot global courant.
        notes (str | None): Notes libres.
        created_at (datetime): Création (local).
        updated_at (datetime | None): MAJ.
    """

    user_id: PyObjectId
    challenge_id: PyObjectId
    # Déclaration UTILISATEUR (peut être "completed" même si non satisfaisant algorithmiquement)
    status: Literal["pending", "accepted", "dismissed", "completed"] = "pending"
    # Statut CALCULÉ par l'évaluation (UCLogic sur les tasks)
    computed_status: Literal["pending", "accepted", "dismissed", "completed"] | None = None
    # Traçabilité de l'override
    manual_override: bool = False
    override_reason: str | None = None
    overridden_at: dt.datetime | None = None
    logic: UCLogic | None = None
    # Aggregated, current snapshot for the whole challenge (redundant with history in Progress
    ↪ collection)
    progress: ProgressSnapshot | None = None
    notes: str | None = None
    # Projection
    estimated_completion_at: dt.datetime | None = None
    created_at: dt.datetime = Field(default_factory=lambda: now())
    updated_at: dt.datetime | None = None
```

6.5 Modèle User Challenge Task

```python
# backend/app/models/user_challenge_task.py
# Tâche déclarée dans un UserChallenge : expression AST, contraintes, statut et métriques.

from __future__ import annotations
import datetime as dt
from pydantic import Field
from app.core.bson_utils import MongoBaseModel, PyObjectId
from app.core.utils import now
from app.models._shared import ProgressSnapshot
from app.models.challenge_ast import TaskExpression

class UserChallengeTask(MongoBaseModel):
    """Document Mongo « UserChallengeTask ».

    Description:
        Contient l'expression AST (sélecteur de caches), les contraintes (ex. min_count),
        le statut manuel, des métriques calculées et un snapshot de progression.

    Attributes:
        user_challenge_id (PyObjectId): Réf. UC parent.
        order (int): Ordre d'affichage.
        title (str): Titre de la tâche.
        expression (TaskExpression): AST de sélection.
        constraints (dict): Contraintes (ex. {'min_count': 4}).
        status (str): 'todo' | 'in_progress' | 'done'.
        metrics (dict): Métriques (ex. {'current_count': 3}).
        progress (ProgressSnapshot | None): Snapshot courant.
        last_evaluated_at (datetime | None): Dernière évaluation.
        created_at (datetime): Création (local).
        updated_at (datetime | None): MAJ.
    """

    user_challenge_id: PyObjectId
    order: int = 0
    title: str
    expression: TaskExpression
    constraints: dict = Field(default_factory=dict)  # ex: {"min_count": 4}
    status: str = Field(default="todo")  # todo | in_progress | done
    metrics: dict = Field(default_factory=dict)  # ex: {"current_count": 3}
    # Current aggregated snapshot for this task (history is in Progress collection)
    progress: ProgressSnapshot | None = None
    start_found_at: dt.datetime | None = None
    completed_at: dt.datetime | None = None

    last_evaluated_at: dt.datetime | None = None
    created_at: dt.datetime = Field(default_factory=lambda: now())
    updated_at: dt.datetime | None = None

UserChallengeTask.model_rebuild()
```

6.6 Modèle Task Expression

```python
# backend/app/models/challenge_ast.py
# AST décrivant les sélecteurs/règles de tâches et la logique (and/or/not) côté UserChallenge.

from __future__ import annotations
from datetime import date
from typing import Any, Literal, Union
from pydantic import BaseModel, ConfigDict, Field
from app.core.bson_utils import PyObjectId

class ASTBase(BaseModel):
    """Base Pydantic pour tous les noeuds AST.
```

```
    Description:
        Active les encoders `PyObjectId` et `populate_by_name`, tolère les types arbitraires,
        afin d'obtenir un JSON/OpenAPI propre pour Swagger.
    """
    model_config = ConfigDict(
        arbitrary_types_allowed=True,
        json_encoders={PyObjectId: str},
        populate_by_name=True,
    )

# ---- Cache-level leaves ----
## --- Selectors ---
class TypeSelector(ASTBase):
    """Sélecteur par type de cache.

    Attributes:
        cache_type_doc_id (PyObjectId | None): Réf. `cache_types._id`.
        cache_type_id (int | None): Identifiant numérique global.
        cache_type_code (str | None): Code type (ex. "whereigo").
    """
    cache_type_doc_id: PyObjectId | None = None
    cache_type_id: int | None = None
    cache_type_code: str | None = Field(
        default=None, description="Cache type code, e.g. 'whereigo'"
    )

# class SizeSelector / StateSelector / CountrySelector / AttributeSelector

## --- Rules ---
class RuleTypeIn(ASTBase):
    """Règle: type ∈ …{}."""
    kind: Literal["type_in"] = "type_in"
    types: list[TypeSelector]

# class RuleSizeIn / RulePlacedYear / RulePlacedBefore / RulePlacedAfter / RuleStateIn /
    ↪ RuleCountryIs / RuleDifficultyBetween / RuleTerrainBetween / RuleAttributes

# ---- Aggregate leaves (apply to the set of eligible finds) ----
class RuleAggSumDifficultyAtLeast(ASTBase):
    """Règle agrégée: somme(difficulté) ≥ min_total (sur l'ensemble de trouvailles éligibles)."""
    kind: Literal["aggregate_sum_difficulty_at_least"] = "aggregate_sum_difficulty_at_least"
    min_total: int = Field(ge=1)

# class RuleAggSumTerrainAtLeast / RuleAggSumDiffPlusTerrAtLeast / RuleAggSumAltitudeAtLeast

TaskLeaf = Union[RuleTypeIn, RuleSizeIn, ..., RuleAggSumDifficultyAtLeast, ...]

class TaskAnd(ASTBase):
    """noeud logique AND.
    Attributes:
        nodes (list[TaskAnd | TaskOr | TaskNot | TaskLeaf]): Sous-noeuds.
    """
    kind: Literal["and"] = "and"
    nodes: list[TaskAnd | TaskOr | TaskNot | TaskLeaf]

# class TaskOr / TaskNot

TaskExpression = TaskAnd | TaskOr | TaskNot | TaskLeaf
TaskAnd.model_rebuild()
TaskOr.model_rebuild()
TaskNot.model_rebuild()


# ---- UC-level logic (composition by task ids, unchanged) ----
class UCAnd(ASTBase):
    """Logique UC: AND des `task_ids`."""
    kind: Literal["and"] = "and"
    task_ids: list[PyObjectId]
```

```
# class UCOr / UCNot

UCLogic = Union[UCAnd, UCOr, UCNot]

# Les kinds logiques et les kinds "feuilles" (règles) connus
_LOGICAL_KINDS = {"and", "or", "not"}
_RULE_KINDS = {"attributes", "type_in", ..., "aggregate_sum_difficulty_at_least",...}

def preprocess_expression_default_and(expr: Any) -> Any:
    """Normalise une expression courte en `AND` explicite.

    Description:
        Transforme les écritures abrégées (sans `kind`, avec règles directes, etc.)
        en une structure canonique où `kind='and'` et les règles sont dans `nodes`.
        Appelée **avant** la validation Pydantic de l'AST.

    Args:
        expr (Any): Expression brute (dict/objets…/).

    Returns:
        Any: Expression normalisée (dict) prête pour la validation.
    """
    # Cas non-dict (list, str, etc.) → inchangé
    if not isinstance(expr, dict):
        return expr

    # Si pas de 'kind' → c'est un AND implicite
    if "kind" not in expr:
        # Si déjà une liste de 'nodes', on force 'and'
        if "nodes" in expr and isinstance(expr["nodes"], list):
            return {"kind": "and", "nodes": expr["nodes"]}

        # Détection d'une "règle courte" (attributs/typage directs)
        looks_like_rule = any(k in expr for k in ("attributes", "type_ids", "codes", "size_ids", "
    ↪ year", "date", "state_ids", "country_id", "min", "max", "min_total"))
        if looks_like_rule:
            return {"kind": "and", "nodes": [expr]}

        # Sinon, on met quand même un AND vide (laisser la validation gérer)
        return {"kind": "and", "nodes": expr.get("nodes", [])}

    # Si 'kind' est une règle au sommet → envelopper dans un AND
    k = expr.get("kind")
    if isinstance(k, str) and k in _RULE_KINDS:
        return {"kind": "and", "nodes": [expr]}

    # Si 'kind' est logique mais sans nodes et qu'on voit des champs de règle,
    # on transforme en nodes=[ ce dict moins 'kind' ] (rare, mais utile)
    if isinstance(k, str) and k in _LOGICAL_KINDS and not expr.get("nodes"):
        looks_like_rule = any(field in expr for field in ("attributes", "type_ids", "codes", "
    ↪ size_ids", "year", "date", "state_ids", "country_id", "min", "max", "min_total"))
        if looks_like_rule:
            rule_like = {kk: vv for kk, vv in expr.items() if kk != "kind"}
            return {"kind": k, "nodes": [rule_like]}

    # Déjà canonique
    return expr
```

7 Autres composants

7.1 Récupération de données d'altimétrie

```
# backend/app/services/providers/elevation_opentopo.py
# Provider OpenTopoData/Mapzen : récupération d'altitudes, découpage des requêtes (URL/compte),
# respect du quota quotidien via la collection `api_quotas`, et rate limiting côté client.

from __future__ import annotations
import asyncio
import os
import httpx
from app.core.settings import get_settings
settings = get_settings()
from app.core.utils import utcnow
from app.db.mongodb import get_collection

# Config
ENDPOINT = settings.elevation_provider_endpoint
MAX_POINTS_PER_REQ = settings.elevation_provider_max_points_per_req
RATE_DELAY_S = settings.elevation_provider_rate_delay_s
URL_MAXLEN = 1800
ENABLED = settings.elevation_enabled

# Quota
PROVIDER_KEY = "opentopodata_mapzen"


def _quota_key_for_today() -> str:
    """Clé de quota journalière pour le provider.

    Description:
        Construit une clé unique pour la journée courante en UTC (via `utcnow()`),
        sous la forme `"opentopodata_mapzen:YYYY-MM-DD"`. Sert d'identifiant de
        document dans la collection `api_quotas`.
    """
    ...


def _read_quota() -> int:
    """Lire le compteur de requêtes du jour."""
    ...

def _inc_quota(n: int) -> None:
    """Incrémenter le compteur de quota du jour."""
    ...

def _build_param(points: list[tuple[float, float]]) -> str:
    """Construire le paramètre `locations` de l'API.

    Description:
        Sérialise la liste de points `(lat, lon)` au format attendu par l'API :
        `"lat,lon|lat,lon|..."`.
    """
    ...


def _split_params_by_url_and_count(all_param: str) -> list[str]:
    """Découper `locations` en fragments compatibles URL et quota par requête."""
    ...

async def fetch(points: list[tuple[float, float]]) -> list[int | None]:
    """Récupérer les altitudes pour une liste de points (alignées sur l'entrée).

    Description:
        - Si le provider est désactivé (`settings.elevation_enabled=False`) **ou** si la liste
          `points` est vide, retourne une liste de `None` de même taille.
        - Respecte un **quota quotidien** en nombre d'appels HTTP, basé sur la collection
```

```
      `api_quotas` et la variable d'environnement `ELEVATION_DAILY_LIMIT` (défaut 1000).
      Si le quota est atteint, retourne des `None` pour les points restants.
    - Construit une chaîne `locations` puis la **découpe** via `_split_params_by_url_and_count`
      en respectant `URL_MAXLEN` et `MAX_POINTS_PER_REQ`.
    - Pour chaque fragment :
        * effectue un `GET` sur `ENDPOINT?locations=...` (timeout configurable par
          `ELEVATION_TIMEOUT_S`, défaut "5.0")
        * parse la réponse JSON et extrait `results[*].elevation`
        * mappe chaque altitude (arrondie à l'entier) au **bon index d'origine**
        * en cas d'erreur HTTP/JSON, laisse les valeurs correspondantes à `None`
        * incrémente le quota et respecte un **rate delay** (`RATE_DELAY_S`) entre appels
          (sauf après le dernier)
    - Ne lève **jamais** d'exception ; toute erreur réseau/parse entraîne des `None` localisés.

    Args:
        points (list[tuple[float, float]]): Liste `(lat, lon)` pour lesquelles obtenir l'altitude.

    Returns:
        list[int | None]: Liste des altitudes en mètres (ou `None` sur échec), **alignée** sur `points`.
    """
    if not ENABLED or not points:
        return [None] * len(points)

    # Respect daily quota (1000 calls/day), counting *requests*, not points
    daily_count = _read_quota()
    DAILY_LIMIT = int(os.getenv("ELEVATION_DAILY_LIMIT", "1000"))
    if daily_count >= DAILY_LIMIT:
        return [None] * len(points)

    # We keep a parallel index list to map back results to original points
    # Build one big param string then split smartly
    param_all = _build_param(points)
    param_chunks = _split_params_by_url_and_count(param_all)
    results: list[int | None] = [None] * len(points)
    # We need to also split the original points list in the same way to keep indices aligned.
    # We'll reconstruct chunk-wise indices by counting commas/pipes.
    idx_start = 0
    async with httpx.AsyncClient(timeout=float(os.getenv("ELEVATION_TIMEOUT_S", "5.0"))) as client:
        for i, param in enumerate(param_chunks):
            # Determine how many points are in this chunk
            n_pts = 1 if param and "|" not in param else (param.count("|") + 1 if param else 0)
            # Quota guard: stop if next request would exceed
            if daily_count >= DAILY_LIMIT:
                break
            url = f"{ENDPOINT}?locations={param}"
            try:
                resp = await client.get(url)
                if resp.status_code == 200:
                    ...
            except Exception:
                pass

            # update quota & delay
            daily_count += 1
            _inc_quota(1)
            idx_start += n_pts

            # Rate-limit (skip after the last chunk)
            if i < len(param_chunks) - 1:
                await asyncio.sleep(RATE_DELAY_S)

    return results
```

8 Copies d'écran Project GC

8.1 Tableau de bord challenges

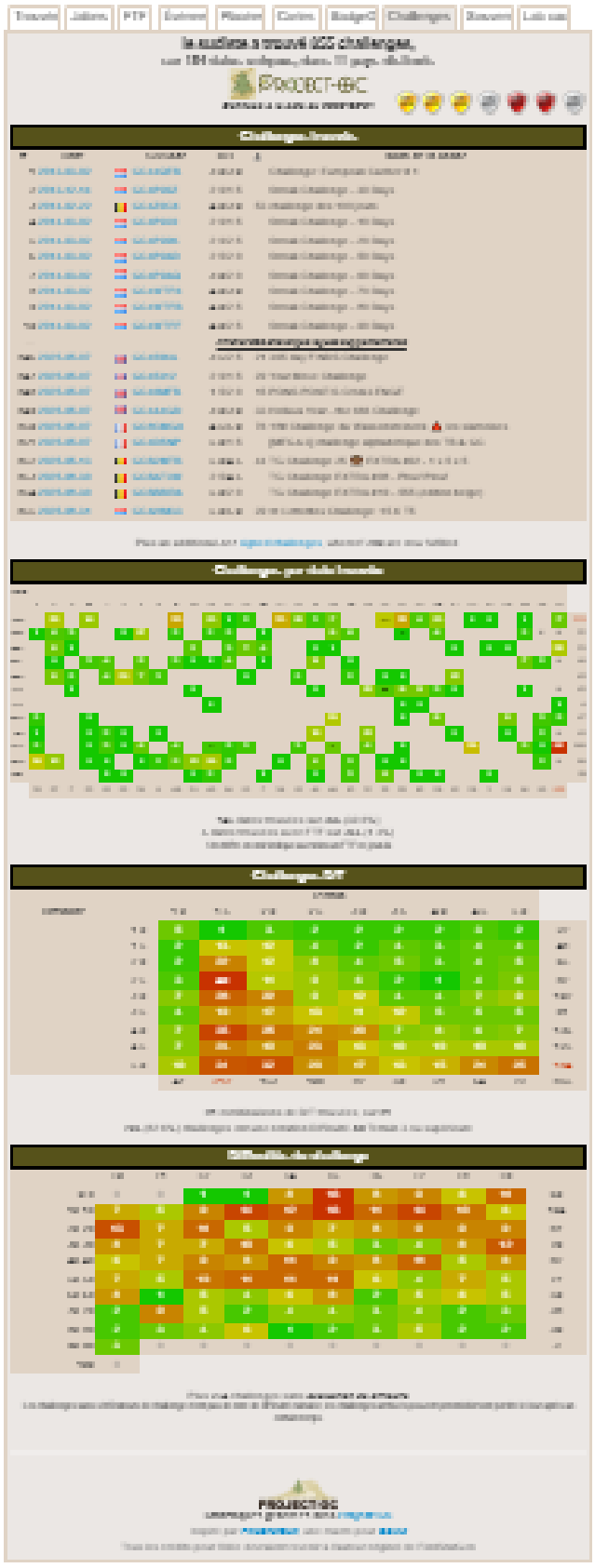


Figure 8 : Tableau de bord challenges

9 Configuration

9.1 Docker Compose

```
# ---------------------------
# docker-compose.yml
# ---------------------------
services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: geo-backend
    ports:
      - "8000:8000"
    env_file:
      - .env
    environment:
      - MONGODB_USER=${MONGODB_USER}
      - MONGODB_PASSWORD=${MONGODB_PASSWORD}
      - MONGODB_URI_TPL=${MONGODB_URI_TPL}
      - MONGODB_DB=${MONGODB_DB}
      - JWT_SECRET_KEY=${JWT_SECRET_KEY}
      - SMTP_HOST=${SMTP_HOST}
      - SMTP_PORT=${SMTP_PORT}
    depends_on:
      - maildev
    volumes:
      - ./backend:/app
      - ./backend/uploads:/app/uploads
      - ./backend/data/samples:/app/data/samples
      - ./.env:/app/.env
    restart: unless-stopped

  maildev:
    container_name: geo-maildev
    image: maildev/maildev
    ports:
      - "1080:1080"
      - "1025:1025"
    restart: unless-stopped

  # --- Frontend unifié avec variable d'environnement ---
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
      target: ${DOCKER_TARGET:-dev}
      args:
        VITE_API_URL: ${VITE_API_URL:-/api}
        VITE_TILE_URL: ${VITE_TILE_URL:-/tiles/{z}/{x}/{y}.png}
    container_name: geo-frontend
    env_file:
      - .env
    environment:
      - CHOKIDAR_USEPOLLING=${CHOKIDAR_USEPOLLING:-true}
      - VITE_API_URL=${VITE_API_URL:-/api}
      - VITE_TILE_URL=${VITE_TILE_URL:-/tiles/{z}/{x}/{y}.png}
    depends_on:
      - backend
      - tiles
    ports:
      - "${FRONTEND_PORT:-5173}:${FRONTEND_INTERNAL_PORT:-5173}"
    volumes:
      - ./frontend:/app
      - /app/node_modules
    restart: unless-stopped
```

```
  # --- Service tiles (commun dev/prod) ---
  tiles:
    image: nginx:1.25-alpine
    container_name: geo-tiles
    restart: unless-stopped
    command: |
      sh -c "
        rm -f /etc/nginx/conf.d/default.conf &&
        mkdir -p /var/log/nginx /var/cache/nginx/tiles_cache &&
        nginx -g 'daemon off;'
      "
    volumes:
      # Configuration optimisée
      - ./ops/nginx/tiles.conf:/etc/nginx/conf.d/tiles.conf:ro
      # Santé + assets locaux
      - ./ops/nginx/www:/var/www:ro
      # Cache persistant avec permissions
      - tiles_cache:/var/cache/nginx/tiles_cache
      # Logs pour debug
      - tiles_logs:/var/log/nginx
    ports:
      - "8080:80"
    # Limite mémoire pour éviter les fuites
    deploy:
      resources:
        limits:
          memory: 512M
        reservations:
          memory: 256M
    # Santé du conteneur
    healthcheck:
      test: ["CMD-SHELL", "wget -q -O /dev/null http://127.0.0.1/tiles/_health.png || exit 1"]
      interval: 30s
      timeout: 3s
      retries: 3
      start_period: 10s

volumes:
  tiles_cache:
    driver: local
  tiles_logs:
    driver: local

# --- Commandes rapides ---
# Développement: docker-compose --profile dev up -d
# Production:    docker-compose --profile prod up -d
# Les deux:      docker-compose --profile dev --profile prod up -d
```

9.2 Backend python

```
aiosmtplib==4.0.2
annotated-types==0.7.0
anyio==4.11.0
bcrypt==4.0.1
black==24.8.0
certifi==2025.8.3
charset-normalizer==3.4.3
click==8.3.0
coverage==7.10.7
dnspython==2.8.0
dotenv==0.9.9
ecdsa==0.19.1
email-validator==2.3.0
fastapi==0.117.1
h11==0.16.0
httpcore==1.0.9
httpx==0.28.1
```

```
idna==3.10
iniconfig==2.1.0
lxml==6.0.2
lxml-stubs==0.5.1
markdown-it-py==4.0.0
mdurl==0.1.2
mypy==1.14.1
mypy_extensions==1.1.0
packaging==25.0
passlib==1.7.4
pathspec==0.12.1
platformdirs==4.4.0
pluggy==1.6.0
pyasn1==0.6.1
pydantic==2.11.9
pydantic-settings==2.10.1
pydantic_core==2.33.2
Pygments==2.19.2
pymongo==4.15.1
pytest==8.3.5
pytest-cov==5.0.0
pytest-env==1.1.5
python-dotenv==1.1.1
python-jose==3.5.0
python-multipart==0.0.20
requests==2.32.5
rich==14.1.0
rsa==4.9.1
ruff==0.13.1
selectolax==0.3.34
six==1.17.0
sniffio==1.3.1
starlette==0.48.0
types-passlib==1.7.7.20250602
types-pyasn1==0.6.0.20250914
types-python-jose==3.5.0.20250531
typing-inspection==0.4.1
typing_extensions==4.15.0
urllib3==2.5.0
uvicorn==0.37.0
```

9.3 Frontend Vue.js

```
{
  "name": "frontend",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "engines": {
    "node": ">=20 <=24"
  },
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "build:test": "vite build --mode test",
    "preview": "vite preview",
    "preview:test": "vite preview --mode test",
    "lint": "eslint . --ext .ts,.tsx,.vue --max-warnings=0",
    "typecheck": "vue-tsc --noEmit",
    "test:unit": "vitest run --coverage --config vitest.config.ts",
    "test:unit:watch": "vitest --config vitest.config.ts",
    "test:e2e": "npm run build:test && playwright test",
    "test:e2e:ui": "npm run build:test && playwright test --ui",
    "test:e2e:headed": "npm run build:test && playwright test --headed",
    "tests:all": "npm run lint && npm run typecheck && npm run test:unit && npm run test:e2e &&
    ↪ npm run build"
  },
```

```
  "dependencies": {
    "@heroicons/vue": "^2.2.0",
    "axios": "^1.11.0",
    "flowbite": "^3.1.2",
    "flowbite-vue": "^0.2.1",
    "leaflet": "^1.9.4",
    "leaflet-draw": "^1.0.4",
    "leaflet.markercluster": "^1.5.3",
    "lucide-vue-next": "^0.542.0",
    "pinia": "^3.0.3",
    "tailwindcss": "^3.4.17",
    "vue": "^3.5.17",
    "vue-router": "^4.5.1",
    "vue-sonner": "^2.0.8"
  },
  "devDependencies": {
    "@eslint/js": "^9.36.0",
    "@playwright/test": "^1.55.1",
    "@types/dompurify": "^3.0.5",
    "@types/leaflet": "^1.9.20",
    "@types/leaflet.markercluster": "^1.5.6",
    "@types/node": "^24.3.0",
    "@typescript-eslint/eslint-plugin": "^8.44.1",
    "@typescript-eslint/parser": "^8.44.1",
    "@vitejs/plugin-vue": "^6.0.1",
    "@vitest/coverage-v8": "^3.2.4",
    "autoprefixer": "^10.4.21",
    "dompurify": "^3.2.7",
    "dotenv": "^17.2.2",
    "eslint": "^9.36.0",
    "eslint-plugin-vue": "^9.33.0",
    "globals": "^16.4.0",
    "jsdom": "^27.0.0",
    "playwright": "^1.47.0",
    "postcss": "^8.5.6",
    "typescript": "^5.9.2",
    "typescript-eslint": "^8.44.1",
    "vite": "^7.0.3",
    "vitest": "^3.2.4",
    "vue-eslint-parser": "^9.4.3",
    "vue-tsc": "^3.0.6"
  },
  "overrides": {
    "esbuild": "^0.25.2"
  }
}
```