Jellyfin → Authentik sur Unraid

migration complète sans changer les mots de passe

État du guide : 15 août 2026

Environnement de référence réellement testé : Unraid, Jellyfin avec base SQLite, plugin officiel LDAP Authentication, Authentik `2026.5.6`, Docker Compose via Compose Manager Plus, Jellyfin en `network_mode=host`.

> **But :** transformer Authentik en fournisseur d'identité central afin qu'un utilisateur puisse conserver un seul compte pour Jellyfin puis, progressivement, pour d'autres applications compatibles LDAP, OIDC/OAuth2, SAML ou Proxy/Forward Auth.

La difficulté n'est pas de faire fonctionner LDAP : la difficulté est de **migrer des comptes Jellyfin existants sans recréer les utilisateurs, sans perdre leurs droits, sans modifier leur identifiant interne et sans imposer un changement de mot de passe**.

La méthode décrite ici provient d'une migration réelle menée jusqu'au bout sur **197 utilisateurs Jellyfin**. Elle a été volontairement construite autour d'audits en lecture seule, de dry-runs, de lots progressifs, d'un rollback automatique et d'une vérification de tous les droits après chaque changement.

1. Ce que l'architecture apporte

```
                AUTHENTIK
            identité centrale
                   |
```

```
          +----------------+----------------+
          |                |                |
         LDAP           OIDC/SAML      Proxy / Forward Auth
          |                |                |
          v                v                v
       Jellyfin        autres apps       apps Web legacy
```

Pour Jellyfin, les clients mobiles, TV et Web continuent à s’authentifier directement auprès de Jellyfin. Jellyfin délègue uniquement la vérification du nom d’utilisateur et du mot de passe au LDAP Authentik.

```
Client Jellyfin
     |
     v
 Jellyfin
     |
 plugin LDAP
     |
     v
 Authentik LDAP Outpost
     |
     v
 Authentik
```

Ne placez pas Jellyfin derrière un Forward Auth Web pour cette architecture : les clients natifs Jellyfin doivent continuer à utiliser normalement l’API Jellyfin. Le Forward Auth est en revanche très utile pour certaines autres applications Web.

2. Ce que la migration doit préserver

- le **Jellyfin User ID** existant ;
- l’historique de lecture, les favoris et toutes les données liées au même compte ;
- les droits d’accès aux bibliothèques ;
- les limitations et paramètres du compte ;
- le statut administrateur défini dans Jellyfin ;
- le mot de passe actuel de l’utilisateur ;
- un compte administrateur Jellyfin local de secours ;
- une possibilité de revenir en arrière pendant la phase de transition.

La règle centrale est donc :

```
On NE supprime PAS l’utilisateur Jellyfin.
On NE recrée PAS l’utilisateur Jellyfin.
On NE modifie PAS ses droits.

On crée l’identité correspondante dans Authentik,
puis on change uniquement :

AuthenticationProviderId : Default -> LDAP
```

3. Comment conserver le mot de passe actuel ?

Sur l’installation qui a servi à valider cette méthode, les mots de passe locaux Jellyfin étaient stockés dans SQLite sous cette forme :

```
$PBKDF2-SHA512$iterations=210000$<salt hex>$<digest hex>
```

Le mot de passe n’est pas récupérable en clair. La solution consiste à importer temporairement le **vérificateur PBKDF2-SHA512** dans Authentik grâce à un hasher Django personnalisé nommé `jf512`.

```
Jellyfin SQLite
$PBKDF2-SHA512$iterations=210000$...
          |
          | conversion d’encodage seulement
          | aucun déchiffrement
          v
Authentik
jf512$210000$SALT_BASE64URL$DIGEST_BASE64URL
          |
          | première connexion réussie
          v
Authentik re-hashe automatiquement le mot de passe
avec son hasher préféré
(dans le setup validé : pbkdf2_sha256)
```

Le hasher `jf512` répond `must_update=True`. Lorsqu'Authentik valide pour la première fois l'ancien mot de passe, son mécanisme normal de vérification remplace automatiquement le hash de transition par le hash préféré d'Authentik.

> **Cette partie est volontairement versionnée et restrictive.** Si votre audit ne retourne pas exactement le format PBKDF2-SHA512 attendu, arrêtez-vous. N'essayez pas d'adapter les scripts au hasard.

4. Pré-requis

Pré-requis	Pourquoi	Remarque
Unraid	Hôte de référence du guide	Les chemins sont écrits pour Unraid
Jellyfin fonctionnel	Serveur existant à migrer	Faire une sauvegarde avant toute modification
Accès administrateur Jellyfin	Création d'une API key, installation du plugin LDAP, gestion du compte de secours	Obligatoire
Clé API Jellyfin	Audits et modification contrôlée des Policies	Ne jamais la mettre dans les scripts ou dans BookStack
Docker Compose	Déploiement d'Authentik	Obligatoire
Compose Manager Plus	Gestion pratique de la stack sur Unraid	Utilisé dans ce guide ; la CLI Docker Compose reste possible
Authentik	Fournisseur d'identité central	Version testée : 2026.5.6
Plugin Jellyfin LDAP Authentication	Authentification Jellyfin contre Authentik	Plugin officiel
Accès terminal Unraid	Scripts, audits, sauvegardes	Obligatoire
`screen`	Ne pas interrompre un batch lors d'une fermeture de terminal	Fortement recommandé
Reverse proxy HTTPS	Accès sécurisé à l'interface Authentik	Nginx Proxy Manager + Let's Encrypt dans le setup testé

4.1. Vérifier le mode réseau Jellyfin

Les commandes réseau de ce guide supposent que Jellyfin est en mode `host`.

```
docker inspect Jellyfin --format 'NetworkMode={{.HostConfig.NetworkMode}}'
```

Résultat attendu pour suivre exactement ce guide :

```
NetworkMode=host
```

Si votre conteneur Jellyfin n’est pas en mode `host`, la migration des comptes reste possible mais la partie réseau LDAP doit être adaptée. Ne copiez pas aveuglément `127.0.0.1:389` dans ce cas.

4.2. Trouver le vrai chemin de la base Jellyfin

Le chemin utilisé pendant la migration réelle était :

```
/mnt/user/appdata/Jellyfin/data/jellyfin.db
```

Votre template peut utiliser une autre casse ou un autre chemin. Inspectez les montages :

```
docker inspect Jellyfin --format '{{range .Mounts}}{{println .Source "->"
.Destination}}{{end}}'
```

Si nécessaire, exportez ensuite le chemin exact :

```
export JF_DB='/mnt/user/appdata/VOTRE_JELLYFIN/data/jellyfin.db'
```

5. Sauvegarder AVANT toute modification

> **Cette étape est obligatoire.** Ne commencez pas par “tester rapidement” un compte de production.

5.1. Jellyfin

Créez une sauvegarde via l’outil de sauvegarde de Jellyfin ou votre stratégie habituelle Unraid. Vérifiez qu’elle est réellement exploitable.

5.2. Authentik PostgreSQL

```
cd /mnt/user/appdata/compose/authentik
mkdir -p backups

docker compose exec -T postgresql \
  pg_dump -U authentik -d authentik -cC \
  > "backups/authentik-pre-jellyfin-migration-$(date +%Y%m%d-%H%M%S).sql"
```

Après installation des scripts, sauvegardez aussi les fichiers spécifiques :

```
cd /mnt/user/appdata/compose/authentik

tar -czf "backups/jellyfin-authentik-migration-files-$(date +%Y%m%d-%H%M%S).tar.gz" \
  data/user_settings.py \
  data/jellyfin_hashers.py \
  data/jellyfin_migration_helper.py \
  migration-audit/
```

Secrets à ne jamais copier dans BookStack, GitHub, un forum ou un log public : clé API Jellyfin, `PG_PASS`, `AUTHENTIK_SECRET_KEY`, token d'Outpost, App Password du compte de service.

6. Installer Authentik avec Docker Compose

La stack de référence est organisée ainsi :

```
/mnt/user/appdata/compose/authentik/
├── compose.yml
├── .env
├── data/
├── certs/
├── custom-templates/
├── backups/
└── migration-audit/
```

Utilisez un **tag de version explicite**. La migration documentée ici a été validée avec :

```
AUTHENTIK_TAG=2026.5.6
```

Exemple de variables :

```
PG_PASS=...
AUTHENTIK_SECRET_KEY=...
AUTHENTIK_TAG=2026.5.6
COMPOSE_PORT_HTTP=9080
COMPOSE_PORT_HTTPS=9443
AUTHENTIK_LDAP_TOKEN=...
```

Depuis Authentik 2025.10, Redis n’est plus requis. Si vous suivez un ancien tutoriel qui ajoute Redis par défaut, ne mélangez pas aveuglément cette architecture avec celle de 2026.5.

Documentation officielle : installation Authentik avec Docker Compose ; notes de version 2025.10 ; notes de version 2026.5.

7. Configurer Authentik pour Jellyfin

7.1. Créer le groupe Jellyfin

Dans **Directory → Groups**, créer :

```
jellyfin_users
```

Ce groupe ne doit pas être Superuser. Il représente uniquement les identités autorisées à apparaître dans la recherche LDAP destinée à Jellyfin.

7.2. Créer l’application

```
Name : Jellyfin
Slug : jellyfin
```

7.3. Créer le LDAP Provider

Name	`Jellyfin LDAP Provider`
Bind flow	`default-authentication-flow` dans le setup testé
Invalidation / Unbind flow	`default-invalidation-flow`
Base DN	`dc=home,dc=lan` — exemple à adapter

Le Base DN n'a pas besoin de correspondre à un vrai domaine DNS. Il doit en revanche être cohérent partout dans la configuration.

Documentation officielle : Create an LDAP provider.

7.4. Créer le compte de service pour le bind

Créer un compte de service, par exemple :

```
jellyfin_service
DN : cn=jellyfin_service,ou=users,dc=home,dc=lan
```

Définir un **App Password** pour ce compte. Jellyfin utilisera ce secret pour le bind LDAP.

> **Ne pas confondre :** l'App Password de `jellyfin_service` sert au bind LDAP ; le token `ak-outpost-…` sert au conteneur Outpost pour communiquer avec Authentik.

7.5. Donner la permission de recherche

Créer un rôle `LDAP Search`, lui associer le compte `jellyfin_service`, puis attribuer sur le Provider la permission :

```
Search full LDAP directory
```

Cette séquence correspond à la procédure Authentik actuelle : compte de service, rôle, permission de recherche, puis Outpost.

8. Déployer l'LDAP Outpost

Créer un Outpost :

```
Name        : Jellyfin LDAP Outpost
Type        : LDAP
Integration : manuel
Application : Jellyfin
```

Stocker le token généré dans `AUTHENTIK_LDAP_TOKEN` du fichier `.env`.

Dans le setup testé, on ajoute ce service directement à la stack Authentik :

```
  authentik_ldap:
    image: ghcr.io/goauthentik/ldap:${AUTHENTIK_TAG}
    restart: unless-stopped
    environment:
      AUTHENTIK_HOST: http://server:9000
      AUTHENTIK_INSECURE: "false"
      AUTHENTIK_TOKEN: ${AUTHENTIK_LDAP_TOKEN}
    ports:
      - "127.0.0.1:389:3389"
```

Pourquoi `127.0.0.1` ? Parce que Jellyfin est en `network_mode=host`. Il partage la pile réseau de l'hôte Unraid et peut donc joindre l'Outpost via le loopback de l'hôte.

```
Jellyfin (host network)
        |
        | 127.0.0.1:389
        v
Unraid loopback
        |
        v
Authentik LDAP Outpost :3389
        |
        | réseau Docker interne de la stack
        v
Authentik server :9000
```

Cette publication ne doit pas apparaître comme `0.0.0.0:389`. Le bind utilisé par Jellyfin reste ainsi local à la machine.

Authentik supporte également LDAPS/StartTLS et le recommande dès que LDAP traverse un réseau. Ici, le LDAP en clair est limité au loopback local ; si vous devez sortir de l'hôte, utilisez TLS.

Documentation officielle : Manual Outpost deployment in Docker Compose et LDAP Provider / LDAPS.

8.1. Vérifier la stack

```
cd /mnt/user/appdata/compose/authentik

docker compose config --quiet
docker compose up -d
docker compose ps

docker ps --format 'table {{.Names}}□{{.Ports}}' | grep ldap
```

Le port doit être publié uniquement sur le loopback :

```
127.0.0.1:389->3389/tcp
```

9. Configurer le plugin LDAP dans Jellyfin

Installer le plugin officiel **LDAP Authentication** depuis le catalogue Jellyfin.

LDAP Server	`127.0.0.1`
LDAP Port	`389`
Secure LDAP	OFF dans ce design loopback local
Bind User	`cn=jellyfin_service,ou=users,dc=home,dc=lan`
Bind Password	App Password Authentik de `jellyfin_service`
Base DN	`dc=home,dc=lan`
Search Filter	`(memberOf=cn=jellyfin_users,ou=groups,dc=home,dc=lan)`
UID Attribute	`uid`
Username Attribute	`cn`
Enable User Creation	**OFF**
LDAP Admin Filter	`_disabled_`

Admin Base DN	vide
Allow Password Change	**OFF** pour cette phase

Plugin officiel : jellyfin-plugin-ldapauth.

9.1. Piège critique : LDAP Admin Filter

> **Si vous voulez conserver Jellyfin comme source de vérité pour les droits administrateur, utilisez `_disabled_`.**

Ce point a été découvert lors des tests réels. Le code du plugin indique que, lorsqu'un Admin Filter est configuré, il recalcule le statut administrateur au login et peut mettre à jour `IsAdministrator` sur un utilisateur existant. Avec `_disabled_`, cette synchronisation est neutralisée.

Source officielle : LDAPAuthenticationProviderPlugin.cs.

9.2. Pourquoi “Enable User Creation” reste OFF

Nous voulons conserver les utilisateurs Jellyfin existants et leur ID. La création automatique d'un nouveau compte par le plugin n'apporte rien à cette migration et introduit un chemin supplémentaire que nous ne voulons pas déclencher.

10. Créer un administrateur Jellyfin local de secours

Avant de migrer un compte réel, créez ou réservez un administrateur qui restera :

```
Authentication Provider : Default
Password Reset Provider  : Default
```

Dans les scripts fournis, le nom protégé par défaut est `admin`. Si votre admin de secours porte un autre nom :

```
export JF_PROTECTED_LOCAL_ADMINS='mon_admin_local'
```

Ne migrez jamais ce compte. C’est votre accès de secours si Authentik ou l’Outpost LDAP est indisponible.

11. Installer le hasher de transition jf512

Copier les deux fichiers suivants dans :

```
/mnt/user/appdata/compose/authentik/data/
```

jellyfin_hashers.py — cliquer pour afficher

```
import base64
import hashlib
```

from django.contrib.auth.hashers import BasePasswordHasher, mask_hash from django.utils.crypto import constant_time_compare from django.utils.translation import gettext_noop as _

def _b64url_encode(raw: bytes) -> str: return base64.urlsafe_b64encode(raw).decode("ascii").rstrip("=")

def _b64url_decode(value: str) -> bytes: return base64.urlsafe_b64decode(value + ("=" * (-len(value) % 4)))

class JellyfinPBKDF2SHA512PasswordHasher(BasePasswordHasher): """Compatibility hasher for Jellyfin PBKDF2-SHA512 local passwords.

```
Imported authentik format:
    jf512$ITERATIONS$SALT_BASE64URL$DIGEST_BASE64URL


must_update() deliberately returns True so a successful login causes
Django/authentik to replace jf512 with the preferred local hasher.
"""


algorithm = "jf512"
```

```
def salt(self):
    raise NotImplementedError(
        "jf512 is an import/verification hasher and must not create new salts"
    )

def encode(self, password, salt, iterations=None):
    if password is None:
        raise TypeError("password must not be None")

    iterations = int(iterations or 210000)
    salt_bytes = _b64url_decode(salt)
    digest = hashlib.pbkdf2_hmac(
        "sha512",
        password.encode("utf-8"),
        salt_bytes,
        iterations,
        dklen=64,
    )
    return f"{self.algorithm}${iterations}${salt}${_b64url_encode(digest)}"

def verify(self, password, encoded):
    try:
        algorithm, iterations, salt, _digest = encoded.split("$", 3)
        if algorithm != self.algorithm:
            return False
        encoded_2 = self.encode(password, salt, int(iterations))
    except (TypeError, ValueError):
        return False
    return constant_time_compare(encoded, encoded_2)

def safe_summary(self, encoded):
    algorithm, iterations, salt, digest = encoded.split("$", 3)
    return {
        _("algorithm"): algorithm,
        _("iterations"): iterations,
        _("salt"): mask_hash(salt),
        _("hash"): mask_hash(digest),
    }
```

```
def must_update(self, encoded):
    return True</code></pre>
```

user_settings.py — cliquer pour afficher

```
"""Custom authentik settings for the Jellyfin password transition.
```

Copy to /data/user_settings.py (host path in this guide: /mnt/user/appdata/compose/authentik/data/user_settings.py).

Do not remove the jf512 hasher while any migrated authentik user still has a password whose algorithm is jf512. """

PASSWORD_HASHERS = ["django.contrib.auth.hashers.PBKDF2PasswordHasher", "data.jellyfin_hashers.JellyfinPBKDF2SHA512PasswordHasher", "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher", "django.contrib.auth.hashers.Argon2PasswordHasher", "django.contrib.auth.hashers.BCryptSHA256PasswordHasher", "django.contrib.auth.hashers.ScryptPasswordHasher",]

Authentik charge `/data/user_settings.py` pendant son démarrage. Le code source Authentik actuel comporte explicitement le chargement de `data.user_settings`.

Source : authentik/root/settings.py.

11.1. Installer les fichiers

```
cd /mnt/user/appdata/compose/authentik

cp /CHEMIN/DU/TOOLKIT/jellyfin_hashers.py data/jellyfin_hashers.py
cp /CHEMIN/DU/TOOLKIT/user_settings.py data/user_settings.py

docker compose restart server worker
```

11.2. Vérifier que le hasher est chargé

```
cd /mnt/user/appdata/compose/authentik

docker compose exec -T server ak shell -c 'from django.conf import settings;
print("\n".join(settings.PASSWORD_HASHERS))'
```

La liste doit contenir :

```
data.jellyfin_hashers.JellyfinPBKDF2SHA512PasswordHasher
```

> **Ne retirez pas ce fichier après le batch.** Il doit rester chargé tant qu'au moins un utilisateur Authentik possède encore un mot de passe `jf512`.

12. Installer le helper Authentik

Placer dans `/mnt/user/appdata/compose/authentik/data/` :

jellyfin_migration_helper.py — cliquer pour afficher

```
import json
import sys
```

from authentik.core.models import Group, User, UserTypes from django.contrib.auth.hashers import identify_hasher

GROUP_NAME = "jellyfin_users" MARKER_KEY = "jellyfin_migration" MARKER_VERSION = 1

def emit(payload): print(json.dumps(payload, ensure_ascii=False, sort_keys=True))

def get_algorithm(user): if not user.password: return "EMPTY" if user.password.startswith("!"): return "UNUSABLE" try: return identify_hasher(user.password).algorithm except Exception: return user.password.split("$", 1)[0]

def exact_and_case_matches(username): exact = User.objects.filter(username=username).first() case_matches = list(User.objects.filter(username__iexact=username).values_list("username", flat=True)) return exact, case_matches

def inspect_user(username): exact, case_matches = exact_and_case_matches(username) if exact is None: return { "ok": True, "exists": False, "username": username, "case_matches":

case_matches, }

```
return {
    "ok": True,
    "exists": True,
    "username": exact.username,
    "uuid": str(exact.uuid),
    "type": exact.type,
    "is_active": exact.is_active,
    "algorithm": get_algorithm(exact),
    "groups": sorted(g.name for g in exact.groups.all()),
    "attributes": exact.attributes,
    "case_matches": case_matches,
}
```

def create_user(username, encoded_hash, jellyfin_id): exact, case_matches = exact_and_case_matches(username)

```
if exact is not None:
    raise RuntimeError(f"User already exists exactly: {username!r}")

if case_matches:
    raise RuntimeError(
        f"Case-insensitive username collision for {username!r}: {case_matches!r}"
    )

group = Group.objects.get(name=GROUP_NAME)
if group.is_superuser:
    raise RuntimeError(f"Refusing to use superuser group {GROUP_NAME!r}")

hasher = identify_hasher(encoded_hash)
if hasher.algorithm != "jf512":
    raise RuntimeError(
        f"Imported password is not recognised as jf512 (got {hasher.algorithm!r})"
    )

marker = {
    MARKER_KEY: {
        "version": MARKER_VERSION,
```

```
        "jellyfin_id": jellyfin_id,
    }
}

user = User.objects.create(
    username=username,
    name=username,
    type=UserTypes.INTERNAL.value,
    is_active=True,
    path="users",
    attributes=marker,
)
user.set_password_from_hash(encoded_hash, signal=False)
user.save()
user.groups.add(group)

result = inspect_user(username)
result["created"] = True
return result
```

def rollback_user(username, jellyfin_id): exact, _case_matches = exact_and_case_matches(username)

```
if exact is None:
    return {
        "ok": True,
        "deleted": False,
        "reason": "already_absent",
        "username": username,
    }

marker = (exact.attributes or {}).get(MARKER_KEY) or {}
if (
    marker.get("version") != MARKER_VERSION
    or marker.get("jellyfin_id") != jellyfin_id
):
    raise RuntimeError(
        "Refusing rollback: authentik account does not carry the expected "
        "migration marker/Jellyfin ID."
```

```
    )

exact.delete()
return {
    "ok": True,
    "deleted": True,
    "username": username,
    "jellyfin_id": jellyfin_id,
}
```

def main(): request = json.loads(sys.stdin.read()) action = request.get("action") username = request.get("username")

```
if not isinstance(username, str):
    raise RuntimeError("username must be a string")

if action == "inspect":
    emit(inspect_user(username))
    return

if action == "create":
    encoded_hash = request.get("encoded_hash")
    jellyfin_id = request.get("jellyfin_id")
    if not isinstance(encoded_hash, str) or not encoded_hash:
        raise RuntimeError("encoded_hash missing")
    if not isinstance(jellyfin_id, str) or not jellyfin_id:
        raise RuntimeError("jellyfin_id missing")
    emit(create_user(username, encoded_hash, jellyfin_id))
    return

if action == "rollback":
    jellyfin_id = request.get("jellyfin_id")
    if not isinstance(jellyfin_id, str) or not jellyfin_id:
        raise RuntimeError("jellyfin_id missing")
    emit(rollback_user(username, jellyfin_id))
    return

raise RuntimeError(f"Unknown action: {action!r}")
```

```
try: main() except Exception as exc: emit({"ok": False, "error": str(exc)}) raise
```

Le helper :

- refuse un username déjà présent ou une collision de casse ;
- exige que `jf512` soit reconnu par Django avant de créer le compte ;
- crée un utilisateur interne Authentik ;
- importe le hash avec `set_password_from_hash(..., signal=False)` ;
- ajoute l'utilisateur à `jellyfin_users` ;
- ajoute un marqueur contenant le Jellyfin ID ;
- n'autorise le rollback que si le marqueur correspond exactement.

Le modèle utilisateur Authentik actuel fournit bien `set_password_from_hash()` et son `check_password()` sait re-hasher automatiquement un mot de passe validé lorsque Django demande une mise à niveau du hash.

Source officielle : authentik/core/models.py.

13. Installer les outils d'audit et de migration

Créer le répertoire :

```
mkdir -p /mnt/user/appdata/compose/authentik/migration-audit
cd /mnt/user/appdata/compose/authentik/migration-audit
```

Copier dans ce répertoire :

- `jellyfin_audit.py`
- `jellyfin_hash_audit.py`
- `cross_audit.py`
- `migrate_one.py`
- `make_batch.py`
- `run_dry_batch.py`
- `run_batch_safe.py`

Copier `audit_authentik_directory.py` dans le dossier `data/` de la stack Authentik.

```
cd /mnt/user/appdata/compose/authentik/migration-audit
chmod +x *.py
```

```
python3 -m py_compile *.py
```

Si `py_compile` ne retourne rien, la syntaxe Python est valide.

14. Créer une clé API Jellyfin

Dans le Dashboard administrateur Jellyfin, ouvrez la section des **API Keys**, créez une clé dédiée à la migration (par exemple `authentik-migration`), puis gardez-la hors des fichiers de documentation.

Dans le terminal, chargez-la sans la mettre en clair dans l'historique shell :

```
cd /mnt/user/appdata/compose/authentik/migration-audit

read -rsp "Clé API Jellyfin : " JF_API
echo
export JF_API

printf 'JF_API length = %s\n' "${#JF_API}"
```

Si votre Jellyfin n'est pas joignable sur `http://127.0.0.1:8096` depuis l'hôte, adaptez :

```
export JF_URL='http://ADRESSE:PORT'
```

15. Audit initial : ne rien migrer tant que l'état n'est pas propre

15.1. Audit Jellyfin et hashes

```
cd /mnt/user/appdata/compose/authentik/migration-audit

./jellyfin_audit.py
./jellyfin_hash_audit.py
```

`jellyfin_hash_audit.py` ouvre SQLite en **mode read-only** et ne sauvegarde jamais le hash complet dans les rapports.

15.2. Audit Authentik

```
cd /mnt/user/appdata/compose/authentik


docker compose exec server ak shell -c
'exec(open("/data/audit_authentik_directory.py").read())'
```

15.3. Croisement Jellyfin ↔ Authentik

```
cd /mnt/user/appdata/compose/authentik/migration-audit
./cross_audit.py
```

15.4. Conditions obligatoires avant de continuer

- le nombre d'utilisateurs API Jellyfin et SQLite doit être identique ;
- `Hashes non migrables : 0` ;
- `Écarts API/DB : 0` ;
- l'admin de secours doit être classé `PROTECTED_LOCAL_ADMIN` ;
- un compte existant dans Authentik ne doit pas entrer en collision avec un compte Jellyfin ;
- **`COMPTES À EXAMINER : Aucun`**.

15.5. Signification des statuts du cross-audit

Statut	Signification
`READY_NEW`	Compte actif local, hash compatible, absent d'Authentik : candidat au batch
`MIGRATED_PENDING_LOGIN`	Compte déjà passé en LDAP, Authentik possède encore le hash de transition jf512
`ALREADY_MIGRATED`	Compte LDAP cohérent et première connexion déjà effectuée ; hash devenu pbkdf2_sha256 dans le setup validé
`READY_NEW_DISABLED`	Compte désactivé : laissé hors batch principal

Statut	Signification
`PROTECTED_LOCAL_ADMIN`	Admin de secours volontairement local
`TEST_ACCOUNT`	Compte pilote / test exclu des batches de production
`CONFLICT_CASE`	Collision de username avec une casse différente
`LDAP_INCONSISTENT`	Jellyfin/Authentik/groupe LDAP ne sont pas cohérents
`LDAP_PASSWORD_STATE_REVIEW`	Algorithme de mot de passe Authentik inattendu
`EXISTING_AUTHENTIK_REVIEW`	Un compte Authentik existe alors que Jellyfin est encore local
`HASH_NOT_MIGRATABLE`	Hash différent du format validé par ce guide
`UNKNOWN_PROVIDER`	Provider Jellyfin inattendu

16. Ne “nettoyez” pas les droits historiques pendant la migration

Un audit peut signaler des IDs de bibliothèques qui ne correspondent plus aux bibliothèques actuelles. Cela peut être un vestige historique. Pendant la migration, **préservez-les** : le but est de garantir que la migration d’authentification ne change aucun droit.

Nettoyer des références historiques est un projet distinct, à faire plus tard.

17. Comprendre le RightsFingerprint

Le fingerprint protège toutes les valeurs `Policy` et `Configuration`, sauf trois champs volontairement exclus du fingerprint de droits :

- `AuthenticationProviderId` : c’est le champ volontairement modifié ;
- `PasswordResetProviderId` : géré dans une phase ultérieure ;
- `InvalidLoginAttemptCount` : compteur volatile de tentatives de login, pas un droit.

`LoginAttemptsBeforeLockout` reste, lui, protégé.

18. Migrer UN compte pilote

Choisissez un compte non critique, dont vous connaissez le mot de passe historique Jellyfin.

18.1. Dry-run

```
./migrate_one.py --user 'utilisateur_test'
```

Le dry-run ne doit faire aucune écriture. Il vérifie le snapshot, le Jellyfin live, le hash SQLite, l’absence du compte Authentik et le statut du cross-audit.

18.2. Apply

```
./migrate_one.py --user 'utilisateur_test' --apply
```

La migration réelle suit quatre étapes :

1. création du compte Authentik avec le hash jf512 et le marqueur Jellyfin ID ;
2. vérification immédiate du compte, du hash et de `jellyfin_users` ;
3. POST de la **Policy Jellyfin complète** en modifiant uniquement `AuthenticationProviderId` ;
4. relecture complète et comparaison exacte des droits.

Le POST complet est volontaire : des chemins Jellyfin exigent notamment `AuthenticationProviderId` et `PasswordResetProviderId` non nuls. Envoyer une Policy partielle est inutilement risqué.

18.3. Test fonctionnel obligatoire

Déconnectez le compte pilote et reconnectez-le à Jellyfin avec **son ancien mot de passe Jellyfin**. Le login doit fonctionner sans reset.

Après ce premier login, relancez l’audit Authentik : l’algorithme du pilote doit passer de `jf512` vers le hasher préféré d’Authentik.

19. PasswordResetProviderId : ne pas le modifier maintenant

```
AuthenticationProviderId : Default -> LDAP
PasswordResetProviderId  : reste Default
```

Authentification et récupération de mot de passe sont deux sujets différents. Les mélanger complique le diagnostic et le rollback. Le workflow de récupération Authentik sera mis en place et testé dans une phase séparée.

Pour la même raison, **Allow Password Change** reste OFF dans le plugin LDAP pendant cette phase.

20. Lancer les batches dans screen

```
screen -S jellyfin-migration


cd /mnt/user/appdata/compose/authentik/migration-audit
read -rsp "Clé API Jellyfin : " JF_API
echo
export JF_API

# Détacher sans interrompre le processus : Ctrl+A puis D
# Revenir ensuite :
screen -r jellyfin-migration
```

> **screen ne survit pas à un reboot ou à une coupure électrique.** Après un redémarrage d'Unraid, refaites les trois audits avant de reprendre. Ne relancez jamais aveuglément un ancien batch.

21. Migrer par paliers

La méthode qui a servi à valider cette procédure a utilisé des lots progressifs. Exemple :

- un compte pilote ;
- petit batch ;
- batch intermédiaire ;
- batch plus important seulement après plusieurs audits propres.

21.1. Créer un batch à partir des READY_NEW

```
./make_batch.py --count 5 --output batch-001.txt

# Visualiser les caractères invisibles / espaces de fin :
sed -n 'l' batch-001.txt
```

21.2. Dry-run de tout le batch

```
./run_dry_batch.py --file batch-001.txt --count 5
```

Il faut obtenir 5/5 dry-runs réussis avant d'appliquer.

21.3. Migration réelle

```
./run_batch_safe.py --file batch-001.txt --count 5
```

Le runner affiche les usernames avec `repr()`, demande une confirmation exacte `MIGRATE 5`, traite les utilisateurs séquentiellement, écrit un log par utilisateur au fil de l'eau et **s'arrête au premier échec**.

21.4. Après CHAQUE batch

```
cd /mnt/user/appdata/compose/authentik/migration-audit
./jellyfin_audit.py
./jellyfin_hash_audit.py

cd /mnt/user/appdata/compose/authentik
docker compose exec server ak shell -c \
'exec(open("/data/audit_authentik_directory.py").read())'

cd /mnt/user/appdata/compose/authentik/migration-audit
./cross_audit.py
```

N'augmentez la taille du lot que si :

- le résumé du batch ne contient aucun FAILED ;
- les compteurs LDAP/Default évoluent comme prévu ;
- le nombre de membres `jellyfin_users` évolue comme prévu ;
- `COMPTES À EXAMINER : Aucun`.

22. Cas réel découvert : username avec espace terminal

Un compte réel avait un username terminé par un espace. La première version du runner utilisait `strip()`. Elle transformait silencieusement :

```
'Fannyvre '  ->  'Fannyvre'
```

Le système de sécurité a arrêté la migration avant toute écriture car le nom exact n'existait plus dans le snapshot.

La correction définitive est essentielle :

```
raw_lines = batch_file.read_text(
    encoding="utf-8"
).splitlines()

# PAS de strip()
users = [
    line
    for line in raw_lines
    if line != ""
]
```

> **Ne normalisez jamais un username pendant une migration.** Préservez exactement casse, accents, espaces initiaux et espaces terminaux.

Utilisez `sed -n 'l'` ou l'affichage `repr()` pour repérer les espaces invisibles.

23. Que faire si un batch s'arrête ?

Ne relancez pas le batch complet. Lisez d’abord le log du compte fautif :

```
cat batch-logs/AAAAmmjj-HHMMSS/NNN-utilisateur.log
cat batch-logs/AAAAmmjj-HHMMSS/summary.txt
```

`migrate_one.py` tente un rollback automatique si l’échec survient après une écriture. Le runner arrête ensuite le batch pour que les comptes suivants ne soient jamais commencés.

Refaites les audits, identifiez l’état réel, corrigez la cause, puis reprenez uniquement les comptes qui n’ont jamais été commencés.

24. Comptes désactivés

Les comptes désactivés sont classés `READY_NEW_DISABLED` et sont volontairement refusés par `migrate_one.py` dans le batch principal.

Décidez séparément s’ils doivent rester locaux, être migrés tout en restant désactivés, ou être archivés. Cette décision ne doit pas contaminer la migration des comptes actifs.

25. Suivre la conversion jf512 → hash Authentik

```
cd /mnt/user/appdata/compose/authentik


docker compose exec server ak shell -c \
'exec(open("/data/audit_authentik_directory.py").read())'
```

Au fil des connexions utilisateurs :

```
MIGRATED_PENDING_LOGIN diminue
ALREADY_MIGRATED augmente
jf512 diminue
pbkdf2_sha256 augmente dans le setup de référence
```

Tant qu’il reste un seul `jf512`, conservez `jellyfin_hashers.py` et `user_settings.py`.

26. Rollback

26.1. Pendant migrate_one.py

Si une étape échoue après création du compte Authentik ou après changement de provider, le script :

1. restaure la Policy Jellyfin complète d’origine ;
2. supprime le compte Authentik créé par la migration uniquement si son marqueur correspond au Jellyfin ID attendu.

26.2. Retour manuel ultérieur

Si vous devez remettre temporairement un utilisateur déjà migré en local :

1. restaurez son `AuthenticationProviderId` vers `Default` via une méthode contrôlée ;
2. vérifiez immédiatement son ID et ses droits ;
3. ne supprimez pas son identité Authentik si elle est déjà utilisée par d’autres applications ;
4. refaites les audits avant toute reprise.

La méthode documentée ne supprime pas le hash local Jellyfin de SQLite, ce qui conserve une porte de retour pendant la transition.

27. Critères de réussite de la phase “authentification”

- `READY_NEW = 0` pour tous les comptes actifs prévus ;
- `COMPTES À EXAMINER : Aucun` ;
- `Hashes non migrables : 0` ;
- `Écarts API/DB : 0` ;
- le compte administrateur de secours est toujours en `Default` ;
- les comptes désactivés sont toujours traités séparément ;
- les comptes migrés sont répartis entre `MIGRATED_PENDING_LOGIN` et `ALREADY_MIGRATED`.

28. Résultat réel ayant validé cette méthode

Mesure	Résultat final de la phase
Utilisateurs Jellyfin	197
Bibliothèques	8
Champs Policy observés	42
Champs Configuration observés	16
Hashes Jellyfin PBKDF2-SHA512 compatibles	197 / 197
Hashes non migrables	0
Écarts API/DB	0
Jellyfin LDAP après migration des actifs	189
Jellyfin Default restant	8
Admin local de secours	1
Comptes désactivés laissés hors batch	7
Utilisateurs Authentik	193
Membres effectifs `jellyfin_users`	189
`MIGRATED_PENDING_LOGIN` à l’audit final	186
`ALREADY_MIGRATED` production à l’audit final	2
`READY_NEW_DISABLED`	7
`PROTECTED_LOCAL_ADMIN`	1
Comptes à examiner	0

Le total LDAP inclut également le compte de test utilisé pendant la validation. Les comptes actifs de production migrés représentaient 188 utilisateurs.

29. Après Jellyfin : un compte Authentik pour plusieurs applications

Une fois l’identité centralisée, choisissez pour chaque application le protocole natif le plus adapté :

- **OIDC/OAuth2** : généralement le meilleur choix pour une application moderne qui le supporte ;
- **SAML** : très courant dans les applications d’entreprise ;
- **LDAP** : utile pour les applications qui attendent un annuaire ;
- **Proxy / Forward Auth** : utile pour certaines applications Web sans authentification exploitable.

Le but n’est pas de forcer LDAP partout : Authentik devient la source d’identité, chaque application utilise le protocole qu’elle sait gérer proprement.

30. Ce qui n’est PAS encore fait par cette phase

- migration du `PasswordResetProviderId` ;
- création d’un flux complet de récupération de mot de passe Authentik ;
- traitement définitif des comptes désactivés ;
- suppression des hashes Jellyfin locaux ;
- retrait du hasher jf512 avant que tous les utilisateurs concernés se soient reconnectés.

31. Commandes de diagnostic courantes

État Authentik

```
cd /mnt/user/appdata/compose/authentik
docker compose ps
docker compose logs --tail=100 server
docker compose logs --tail=100 authentik_ldap
```

Port LDAP

```
docker ps --format 'table {{.Names}}□{{.Ports}}' | grep ldap
```

Mode réseau Jellyfin

```
docker inspect Jellyfin --format 'NetworkMode={{.HostConfig.NetworkMode}}'
```

Dry-run d’un utilisateur exact

```
./migrate_one.py --user 'nom exact'
```

Inspecter un utilisateur dans Authentik

```
cd /mnt/user/appdata/compose/authentik

printf '%s\n' '{"action":"inspect","username":"nom exact"}' | \
docker compose exec -T server ak shell -c \
'exec(open("/data/jellyfin_migration_helper.py").read())'
```

32. Limites, maintenance et compatibilité

> **Ne considérez pas ce guide comme un protocole universel pour toutes les versions futures.**

- Authentik et Jellyfin évoluent ;
- le format du hash Jellyfin peut changer ;
- le schéma SQLite peut changer ;
- le nom interne du provider LDAP peut changer ;
- les champs de Policy peuvent évoluer ;
- le comportement du plugin LDAP peut évoluer.

Après une mise à jour majeure, refaites d’abord les audits sur une copie/sauvegarde et comparez le format de hash, le schéma et les noms de providers avant toute nouvelle migration.

33. Checklist avant le premier apply

1. Backup Jellyfin validé.
2. Backup PostgreSQL Authentik validé.
3. Authentik et LDAP Outpost démarrent correctement.
4. LDAP exposé seulement sur `127.0.0.1:389` dans le design host-mode.
5. Bind `jellyfin_service` fonctionne.
6. Groupe `jellyfin_users` non Superuser.
7. Jellyfin LDAP User Creation = OFF.
8. LDAP Admin Filter = `_disabled_`.
9. Allow Password Change = OFF.
10. Admin Jellyfin local de secours présent et testé.
11. Hasher `jf512` chargé par Authentik.
12. API key Jellyfin chargée uniquement en variable d’environnement.
13. Hash audit : 0 non migrable.
14. API/DB : 0 écart.
15. Cross-audit : aucun compte à examiner.
16. Compte pilote identifié.
17. Dry-run du pilote réussi.

34. Fichiers complets fournis avec ce guide

L’archive compagnon contient tous les scripts ci-dessous. Ils sont également reproduits ici pour que la page BookStack reste auto-suffisante.

jellyfin_audit.py — cliquer pour afficher

```
#!/usr/bin/env python3
import csv
import datetime as dt
```

```
import hashlib
import json
import os
import urllib.request
from pathlib import Path
```

JF_URL = os.environ.get("JF_URL", "http://127.0.0.1:8096").rstrip("/") JF_API = os.environ.get("JF_API", "") ROOT = Path(**file**).resolve().parent SNAPSHOTS = ROOT / "snapshots"

Volatile/authentication fields intentionally excluded from the rights-only fingerprint. Everything else in Policy plus all Configuration is protected.

RIGHTS_POLICY_EXCLUDE = { "AuthenticationProviderId", "PasswordResetProviderId", "InvalidLoginAttemptCount", }

def stamp(): return dt.datetime.now().strftime("%Y%m%d-%H%M%S")

def api_get(path): if not JF_API: raise SystemExit(" JF_API n'est pas exportée.")

```
req = urllib.request.Request(
    f"{JF_URL}{path}",
    headers={
```

```
        "Accept": "application/json",
        "Authorization": (
            'MediaBrowser '
            f'Token="{JF_API}", '
            'Client="jellyfin-authentik-audit", '
            'Device="Unraid", '
            'DeviceId="jellyfin-authentik-audit", '
            'Version="1.0"'
        ),
    },
)
with urllib.request.urlopen(req, timeout=30) as response:
    return json.load(response)
```

def fingerprint(obj): raw = json.dumps(obj, ensure_ascii=False, sort_keys=True, separators=(",", ":"),).encode("utf-8") return hashlib.sha256(raw).hexdigest()

def rights_fingerprint(user): policy = dict(user.get("Policy") or {}) for key in RIGHTS_POLICY_EXCLUDE: policy.pop(key, None) return fingerprint({ "Policy": policy, "Configuration": user.get("Configuration") or {}, })

def full_fingerprint(user): return fingerprint({ "Policy": user.get("Policy") or {}, "Configuration": user.get("Configuration") or {}, })

def main(): out = SNAPSHOTS / stamp() out.mkdir(parents=True, exist_ok=False)

```
print("=" * 60)
print(" AUDIT JELLYFIN → AUTHENTIK")
print(" MODE : LECTURE SEULE")
print("=" * 60)
print()
print("Serveur :", JF_URL)
print("Dossier :", out.relative_to(ROOT))
print()

print("Lecture des utilisateurs...")
user_index = api_get("/Users")
users = []
for item in user_index:
    user = api_get(f"/Users/{item['Id']}")
    user["RightsFingerprint"] = rights_fingerprint(user)
```

```
    user["FullFingerprint"] = full_fingerprint(user)
    users.append(user)

print("Lecture des bibliothèques...")
folders = api_get("/Library/VirtualFolders")

print("Lecture des informations serveur...")
system = api_get("/System/Info")

policy_fields = set()
config_fields = set()
admin_count = 0
disabled_count = 0

# Preserve unresolved IDs rather than cleaning them. VirtualFolder ItemId is
# the best available reference for the library access IDs in the tested setup.
known_folder_ids = {
    f.get("ItemId") for f in folders if isinstance(f, dict) and f.get("ItemId")
}
unresolved_ids = set()

for user in users:
    policy = user.get("Policy") or {}
    config = user.get("Configuration") or {}
    policy_fields.update(policy.keys())
    config_fields.update(config.keys())

    if policy.get("IsAdministrator"):
        admin_count += 1
    if policy.get("IsDisabled"):
        disabled_count += 1

    for field in ("EnabledFolders", "BlockedMediaFolders"):
        for folder_id in policy.get(field) or []:
            if folder_id and folder_id not in known_folder_ids:
                unresolved_ids.add(folder_id)

payload = {
    "created_at": out.name,
```

```
    "server": JF_URL,
    "system": system,
    "folders": folders,
    "users": users,
    "summary": {
        "users": len(users),
        "libraries": len(folders),
        "policy_field_count": len(policy_fields),
        "configuration_field_count": len(config_fields),
        "admins": admin_count,
        "disabled": disabled_count,
        "unresolved_library_ids": sorted(unresolved_ids),
    },
}

(out / "jellyfin-audit.json").write_text(
    json.dumps(payload, indent=2, ensure_ascii=False),
    encoding="utf-8",
)
(out / "jellyfin-users-raw.json").write_text(
    json.dumps(users, indent=2, ensure_ascii=False),
    encoding="utf-8",
)
(out / "jellyfin-virtual-folders-raw.json").write_text(
    json.dumps(folders, indent=2, ensure_ascii=False),
    encoding="utf-8",
)
(out / "jellyfin-system-info-raw.json").write_text(
    json.dumps(system, indent=2, ensure_ascii=False),
    encoding="utf-8",
)

csv_fields = [
    "Name", "Id", "IsAdministrator", "IsDisabled",
    "AuthenticationProviderId", "PasswordResetProviderId",
    "RightsFingerprint", "FullFingerprint",
]
with (out / "jellyfin-audit.csv").open("w", encoding="utf-8", newline="") as fh:
    writer = csv.DictWriter(fh, fieldnames=csv_fields)
```

```
    writer.writeheader()
    for u in users:
        p = u.get("Policy") or {}
        writer.writerow(
            {
                "Name": u.get("Name"),
                "Id": u.get("Id"),
                "IsAdministrator": p.get("IsAdministrator"),
                "IsDisabled": p.get("IsDisabled"),
                "AuthenticationProviderId": p.get("AuthenticationProviderId"),
                "PasswordResetProviderId": p.get("PasswordResetProviderId"),
                "RightsFingerprint": u.get("RightsFingerprint"),
                "FullFingerprint": u.get("FullFingerprint"),
            }
        )

summary_lines = [
    f"Utilisateurs : {len(users)}",
    f"Bibliothèques : {len(folders)}",
    f"Policy :        {len(policy_fields)} champs distincts",
    f"Configuration: {len(config_fields)} champs distincts",
    f"IDs bibliothèque non résolus : {len(unresolved_ids)}",
    f"Administrateurs : {admin_count}",
    f"Comptes désactivés : {disabled_count}",
]
(out / "summary.txt").write_text("\n".join(summary_lines) + "\n", encoding="utf-8")

print()
print("□ Audit terminé.")
print()
for line in summary_lines:
    print(line)
print()
print("Fichiers produits :")
for name in (
    "jellyfin-audit.csv", "jellyfin-audit.json", "jellyfin-system-info-raw.json",
    "jellyfin-users-raw.json", "jellyfin-virtual-folders-raw.json", "summary.txt"
):
    print(" -", out.relative_to(ROOT) / name)
```

```
print()
print("□□AUCUNE ÉCRITURE N'A ÉTÉ EFFECTUÉE SUR JELLYFIN.")
```

if **name** == "**main**": main()

jellyfin_hash_audit.py — cliquer pour afficher

```
#!/usr/bin/env python3
import json
import os
import re
import sqlite3
from pathlib import Path
```

ROOT = Path(**file**).resolve().parent DB_PATH = Path(os.environ.get("JF_DB", "/mnt/user/appdata/Jellyfin/data/jellyfin.db"))

DEFAULT_PROVIDER = "Jellyfin.Server.Implementations.Users.DefaultAuthenticationProvider" LDAP_PROVIDER = os.environ.get("JF_LDAP_PROVIDER", "Jellyfin.Plugin.LDAP_Auth.LdapAuthenticationProviderPlugin",)

HASH_RE = re.compile(r"^$PBKDF2-SHA512$iterations=(\d+)$([0-9A-Fa-f]+)$([0-9A-Fa-f]+)$")

def latest_snapshot(): root = ROOT / "snapshots" dirs = sorted(p for p in root.iterdir() if p.is_dir()) if root.exists() else [] if not dirs: raise SystemExit("□ Aucun snapshot. Lancez d'abord ./jellyfin_audit.py") return dirs[-1]

def inspect_hash(value): if not isinstance(value, str) or not value: return {"format": "MISSING", "migratable": False, "reason": "missing"} m = HASH_RE.match(value) if not m: return {"format": "OTHER", "migratable": False, "reason": "unexpected_format"}

```
iterations = int(m.group(1))
salt_hex = m.group(2)
digest_hex = m.group(3)

if len(salt_hex) != 32:
    return {
        "format": "PBKDF2-SHA512", "iterations": iterations,
        "migratable": False, "reason": f"salt_bytes={len(salt_hex)//2}",
    }
```

```
if len(digest_hex) != 128:
    return {
        "format": "PBKDF2-SHA512", "iterations": iterations,
        "migratable": False, "reason": f"digest_bytes={len(digest_hex)//2}",
    }
if iterations != 210000:
    return {
        "format": "PBKDF2-SHA512", "iterations": iterations,
        "migratable": False,
        "reason": "iteration_count_not_validated_by_this_guide",
    }

return {
    "format": "PBKDF2-SHA512",
    "iterations": iterations,
    "migratable": True,
    "reason": "ok",
}
```

def main(): snap = latest_snapshot() audit = json.loads((snap / "jellyfin-audit.json").read_text(encoding="utf-8")) api_by_id = {u["Id"]: u for u in audit["users"]}

```
uri = f"file:{DB_PATH}?mode=ro"
conn = sqlite3.connect(uri, uri=True)
conn.row_factory = sqlite3.Row
try:
    rows = conn.execute(
        """
        SELECT Id, Username, Password, AuthenticationProviderId, PasswordResetProviderId
        FROM Users
        """
    ).fetchall()
finally:
    conn.close()

records = []
formats = {}
ldap_count = 0
default_count = 0
```

```
non_migratable = 0
discrepancies = []
db_ids = set()

for row in rows:
    user_id = row["Id"]
    db_ids.add(user_id)
    parsed = inspect_hash(row["Password"])
    formats[parsed["format"]] = formats.get(parsed["format"], 0) + 1
    non_migratable += int(not parsed["migratable"])

    provider = row["AuthenticationProviderId"]
    if provider == LDAP_PROVIDER:
        ldap_count += 1
    elif provider == DEFAULT_PROVIDER:
        default_count += 1

    api_user = api_by_id.get(user_id)
    if api_user is None:
        discrepancies.append({"id": user_id, "username": row["Username"], "reason":
"db_only"})
    elif api_user.get("Name") != row["Username"]:
        discrepancies.append(
            {
                "id": user_id,
                "db_username": row["Username"],
                "api_username": api_user.get("Name"),
                "reason": "username_mismatch",
            }
        )

    # Intentionally never save the full password hash.
    records.append(
        {
            "id": user_id,
            "username": row["Username"],
            "authentication_provider": provider,
            "password_reset_provider": row["PasswordResetProviderId"],
            **parsed,
```

```
        }
    )

for user_id, api_user in api_by_id.items():
    if user_id not in db_ids:
        discrepancies.append({"id": user_id, "username": api_user.get("Name"), "reason":
"api_only"})

admins_default = sum(
    1
    for u in audit["users"]
    if (u.get("Policy") or {}).get("IsAdministrator")
    and (u.get("Policy") or {}).get("AuthenticationProviderId") == DEFAULT_PROVIDER
)

payload = {
    "database": str(DB_PATH),
    "snapshot": snap.name,
    "users_api": len(api_by_id),
    "users_db": len(rows),
    "formats": formats,
    "ldap": ldap_count,
    "default": default_count,
    "admins_still_default": admins_default,
    "non_migratable": non_migratable,
    "api_db_discrepancies": discrepancies,
    "users": records,
}
(snap / "jellyfin-hash-audit.json").write_text(
    json.dumps(payload, indent=2, ensure_ascii=False), encoding="utf-8"
)

lines = [
    f"Utilisateurs API : {len(api_by_id)}",
    f"Utilisateurs DB  : {len(rows)}",
    "",
    "Formats :",
]
for fmt, count in sorted(formats.items()):
```

```
    lines.append(f"  {fmt:&lt;28} {count}")
lines += [
    "",
    f"LDAP : {ldap_count}",
    f"Default : {default_count}",
    "",
    f"Admins encore en Default : {admins_default}",
    f"Hashes non migrables : {non_migratable}",
    f"Écarts API/DB : {len(discrepancies)}",
]
(snap / "hash-summary.txt").write_text("\n".join(lines) + "\n", encoding="utf-8")

print("=" * 60)
print(" AUDIT HASHES JELLYFIN")
print(" MODE : LECTURE SEULE")
print("=" * 60)
print()
print("Base :", DB_PATH)
print("Snapshot :", snap.relative_to(ROOT))
print()
for line in lines:
    print(line)
print()
print("□ Rapport :", snap.relative_to(ROOT) / "hash-summary.txt")
print()
print("□□Aucun hash complet sauvegardé.")
print("□□Aucune écriture SQLite.")
```

if **name** == "**main**": main()

audit_authentik_directory.py — cliquer pour afficher

```
import datetime as dt
import json
from pathlib import Path
```

from authentik.core.models import Group, User from django.contrib.auth.hashers import identify_hasher

GROUP_NAME = "jellyfin_users" OUT_ROOT = Path("/data/migration-audit")

def password_algorithm(user): value = user.password or "" if not value: return "EMPTY" if value.startswith("!"): return "UNUSABLE" try: return identify_hasher(value).algorithm except Exception: return value.split("$", 1)[0]

def main(): stamp = dt.datetime.now().strftime("%Y%m%d-%H%M%S") out = OUT_ROOT / stamp out.mkdir(parents=True, exist_ok=False)

```
users = list(User.objects.all().order_by("username"))
groups = list(Group.objects.all().order_by("name"))
group = Group.objects.filter(name=GROUP_NAME).first()

algorithms = {}
type_counts = {}
records = []

for user in users:
    algo = password_algorithm(user)
    algorithms[algo] = algorithms.get(algo, 0) + 1
    type_counts[user.type] = type_counts.get(user.type, 0) + 1
    direct_groups = sorted(g.name for g in user.groups.all())
    effective_groups = sorted(g.name for g in user.all_groups())
    records.append(
        {
            "username": user.username,
            "uuid": str(user.uuid),
            "type": user.type,
            "is_active": user.is_active,
            "is_superuser": user.is_superuser,
            "algorithm": algo,
            "direct_groups": direct_groups,
            "effective_groups": effective_groups,
            "attributes": user.attributes,
        }
    )

direct_members = []
effective_members = []
parents = []
```

```
roles = []

if group is not None:
    direct_members = sorted(
        User.objects.filter(groups=group).values_list("username", flat=True)
    )
    effective_members = sorted(
        user.username
        for user in users
        if group.name in {g.name for g in user.all_groups()}
    )

    parent = getattr(group, "parent", None)
    if parent is not None:
        try:
            parents = [parent.name] if parent else []
        except Exception:
            parents = []

    try:
        roles = sorted(role.name for role in group.roles.all())
    except Exception:
        roles = []

payload = {
    "created_at": stamp,
    "summary": {
        "users": len(users),
        "groups": len(groups),
        "types": type_counts,
        "algorithms": algorithms,
    },
    "group": None if group is None else {
        "name": group.name,
        "is_superuser": group.is_superuser,
        "direct_members": direct_members,
        "effective_members": effective_members,
        "parents": parents,
        "roles": roles,
```

```
    },
    "users": records,
}

(out / "authentik-directory-audit.json").write_text(
    json.dumps(payload, indent=2, ensure_ascii=False), encoding="utf-8"
)

print("AUDIT ANNUAIRE AUTHENTIK")
print("=" * 70)
print()
print("MODE : LECTURE SEULE")
print()
print("Utilisateurs :", len(users))
print("Groupes      :", len(groups))
print()
print("TYPES D'UTILISATEURS")
print("-" * 70)
for type_name, count in sorted(type_counts.items()):
    print(f"{count:4d}  {type_name}")

print()
print("GROUPE", GROUP_NAME)
print("-" * 70)
if group is None:
    print("Présent                 : non")
else:
    print("Présent                 : oui")
    print("Superuser               :", group.is_superuser)
    print("Membres directs         :", len(direct_members))
    print("Membres effectifs       :", len(effective_members))
    print("Parents                 :", ", ".join(parents) if parents else "aucun")
    print("Rôles                   :", ", ".join(roles) if roles else "aucun")

print()
print("MEMBRES EFFECTIFS", GROUP_NAME)
print("-" * 70)
for username in effective_members:
    print(" -", username)
```

```
print()
print("SUPERUSERS AUTHENTIK")
print("-" * 70)
supers = [u for u in users if u.is_superuser]
if not supers:
    print(" Aucun")
else:
    for user in supers:
        print(" -", user.username)

print()
print("ALGORITHMES DE MOT DE PASSE")
print("-" * 70)
for algo, count in sorted(algorithms.items()):
    print(f"{count:4d}  {algo}")

print()
print("AUCUNE MODIFICATION AUTHENTIK EFFECTUÉE.")
print("AUCUN HASH DE MOT DE PASSE SAUVEGARDÉ.")
print()
print("JSON :", out / "authentik-directory-audit.json")
```

main()

cross_audit.py — cliquer pour afficher

```
#!/usr/bin/env python3
import csv
import datetime as dt
import json
import os
from pathlib import Path
```

ROOT = Path(**file**).resolve().parent AUTH_AUDIT_ROOT = Path(os.environ.get("AUTH_AUDIT_ROOT", str(ROOT.parent / "data" / "migration-audit"),))

DEFAULT_PROVIDER = "Jellyfin.Server.Implementations.Users.DefaultAuthenticationProvider" LDAP_PROVIDER = os.environ.get("JF_LDAP_PROVIDER",

"Jellyfin.Plugin.LDAP_Auth.LdapAuthenticationProviderPlugin",)

PROTECTED = { value for value in os.environ.get("JF_PROTECTED_LOCAL_ADMINS", "admin").split(",") if value } TEST_ACCOUNTS = { value for value in os.environ.get("JF_TEST_ACCOUNTS", "hashtest").split(",") if value }

PROBLEM_STATUSES = { "CONFLICT_CASE", "LDAP_PASSWORD_STATE_REVIEW", "LDAP_INCONSISTENT", "EXISTING_AUTHENTIK_REVIEW", "HASH_NOT_MIGRATABLE", "UNKNOWN_PROVIDER", }

def latest_dir(root): dirs = sorted(p for p in root.iterdir() if p.is_dir()) if root.exists() else [] if not dirs: raise SystemExit(f"□ Aucun audit trouvé dans {root}") return dirs[-1]

def main(): jf_dir = latest_dir(ROOT / "snapshots") auth_dir = latest_dir(AUTH_AUDIT_ROOT)

```
jf = json.loads((jf_dir / "jellyfin-audit.json").read_text(encoding="utf-8"))
hashes = json.loads((jf_dir / "jellyfin-hash-audit.json").read_text(encoding="utf-8"))
auth = json.loads((auth_dir / "authentik-directory-audit.json").read_text(encoding="utf-
8"))

hash_by_id = {u["id"]: u for u in hashes["users"]}
auth_by_exact = {u["username"]: u for u in auth["users"]}
auth_casefold = {}
for u in auth["users"]:
    auth_casefold.setdefault(u["username"].casefold(), []).append(u["username"])

rows = []

for user in jf["users"]:
    username = user["Name"]
    user_id = user["Id"]
    policy = user.get("Policy") or {}
    is_admin = bool(policy.get("IsAdministrator"))
    is_disabled = bool(policy.get("IsDisabled"))
    provider = policy.get("AuthenticationProviderId")
    reset_provider = policy.get("PasswordResetProviderId")
    h = hash_by_id.get(user_id) or {}
    auth_user = auth_by_exact.get(username)
    case_matches = [
        name for name in auth_casefold.get(username.casefold(), []) if name != username
    ]
```

```
    reasons = []

    if username in PROTECTED:
        status = "PROTECTED_LOCAL_ADMIN"
        reasons.append("Compte explicitement protégé et conservé en authentification
locale.")

    elif username in TEST_ACCOUNTS:
        status = "TEST_ACCOUNT"
        reasons.append("Compte de test explicitement exclu des batches de production.")

    elif case_matches and auth_user is None:
        status = "CONFLICT_CASE"
        reasons.append(
            "Un compte Authentik existe avec une casse différente : "
            + ", ".join(repr(x) for x in case_matches)
        )

    elif provider == LDAP_PROVIDER:
        if auth_user is None:
            status = "LDAP_INCONSISTENT"
            reasons.append("Jellyfin est LDAP mais aucun compte Authentik exact
n'existe.")
        elif "jellyfin_users" not in auth_user.get("effective_groups", []):
            status = "LDAP_INCONSISTENT"
            reasons.append("Compte Authentik absent du groupe effectif jellyfin_users.")
        elif auth_user.get("algorithm") == "jf512":
            status = "MIGRATED_PENDING_LOGIN"
            reasons.append("Migration technique terminée ; première connexion encore
attendue.")
        elif auth_user.get("algorithm") == "pbkdf2_sha256":
            status = "ALREADY_MIGRATED"
            reasons.append("Compte LDAP cohérent et mot de passe déjà re-hashé par
Authentik.")
        else:
            status = "LDAP_PASSWORD_STATE_REVIEW"
            reasons.append(
                "Compte LDAP cohérent mais algorithme de mot de passe inattendu : "
                f"{auth_user.get('algorithm')!r}"
```

```
            )

    elif provider == DEFAULT_PROVIDER:
        if is_disabled:
            if auth_user is None and h.get("migratable"):
                status = "READY_NEW_DISABLED"
                reasons.append("Compte désactivé, techniquement migrable, laissé hors
batch.")
            elif auth_user is not None:
                status = "EXISTING_AUTHENTIK_REVIEW"
                reasons.append("Compte désactivé local mais déjà présent dans Authentik.")
            else:
                status = "HASH_NOT_MIGRATABLE"
                reasons.append("Compte désactivé avec hash non migrable.")
        elif auth_user is not None:
            status = "EXISTING_AUTHENTIK_REVIEW"
            reasons.append(
                "Compte encore local dans Jellyfin mais username exact déjà présent dans
Authentik."
            )
        elif not h.get("migratable"):
            status = "HASH_NOT_MIGRATABLE"
            reasons.append(h.get("reason", "hash non migrable"))
        else:
            status = "READY_NEW"
            reasons.append("Compte local Jellyfin, hash compatible et absent
d'Authentik.")

    else:
        status = "UNKNOWN_PROVIDER"
        reasons.append(f"Provider Jellyfin inattendu : {provider!r}")

    rows.append(
        {
            "username": username,
            "jellyfin_id": user_id,
            "status": status,
            "reasons": reasons,
            "is_admin": is_admin,
```

```
            "is_disabled": is_disabled,
            "authentication_provider": provider,
            "password_reset_provider": reset_provider,
            "hash_migratable": bool(h.get("migratable")),
            "rights_fingerprint": user.get("RightsFingerprint"),
            "full_fingerprint": user.get("FullFingerprint"),
            "authentik_exists": auth_user is not None,
            "authentik_type": None if auth_user is None else auth_user.get("type"),
            "authentik_password_algorithm": None if auth_user is None else
auth_user.get("algorithm"),
            "authentik_effective_groups": [] if auth_user is None else
auth_user.get("effective_groups", []),
            "authentik_in_jellyfin_users": False if auth_user is None else
"jellyfin_users" in auth_user.get("effective_groups", []),
            "case_conflict": case_matches or None,
        }
    )

out_root = ROOT / "cross-audits"
out_root.mkdir(exist_ok=True)
out = out_root / dt.datetime.now().strftime("%Y%m%d-%H%M%S")
suffix = 1
while out.exists():
    out = out_root / f"{out.name}-{suffix:02d}"
    suffix += 1
out.mkdir()

payload = {
    "created_at": out.name,
    "jellyfin_snapshot": jf_dir.name,
    "authentik_snapshot": auth_dir.name,
    "jellyfin_users": len(jf["users"]),
    "authentik_users": len(auth["users"]),
    "users": rows,
}
(out / "cross-audit.json").write_text(
    json.dumps(payload, indent=2, ensure_ascii=False), encoding="utf-8"
)
```

```
fieldnames = [
    "username", "jellyfin_id", "status", "is_admin", "is_disabled",
    "authentication_provider", "password_reset_provider", "hash_migratable",
    "rights_fingerprint", "authentik_exists", "authentik_password_algorithm",
    "authentik_in_jellyfin_users", "case_conflict",
]
with (out / "cross-audit.csv").open("w", newline="", encoding="utf-8") as fh:
    writer = csv.DictWriter(fh, fieldnames=fieldnames)
    writer.writeheader()
    for row in rows:
        flat = dict(row)
        flat["case_conflict"] = json.dumps(row["case_conflict"], ensure_ascii=False)
        writer.writerow({k: flat.get(k) for k in fieldnames})

counts = {}
for row in rows:
    counts[row["status"]] = counts.get(row["status"], 0) + 1
problems = [row for row in rows if row["status"] in PROBLEM_STATUSES]

def section(title, selected):
    print()
    print(title)
    print("-" * 70)
    if not selected:
        print(" Aucun")
    else:
        for row in selected:
            print(" -", row["username"])

print("CROISEMENT JELLYFIN ↔ AUTHENTIK")
print("=" * 70)
print()
print("MODE : LECTURE SEULE")
print()
print("Utilisateurs Jellyfin :", len(jf["users"]))
print("Utilisateurs Authentik :", len(auth["users"]))
print()
print("STATUTS")
print("-" * 70)
```

```
for status in sorted(counts):
    print(f"{counts[status]:4d}  {status}")

print()
print("COMPTES À EXAMINER")
print("-" * 70)
if not problems:
    print(" Aucun")
else:
    for row in problems:
        print(f" - {row['username']!r}: {row['status']}")
        for reason in row["reasons"]:
            print("   ", reason)

section("COMPTES PROTÉGÉS", [r for r in rows if r["status"] == "PROTECTED_LOCAL_ADMIN"])
section("COMPTES TEST", [r for r in rows if r["status"] == "TEST_ACCOUNT"])
section("DÉJÀ MIGRÉS", [r for r in rows if r["status"] == "ALREADY_MIGRATED"])
section("COMPTES DÉSACTIVÉS MAIS MIGRABLES", [r for r in rows if r["status"] ==
"READY_NEW_DISABLED"])

jellyfin_names = {r["username"] for r in rows}
extras = [u for u in auth["users"] if u["username"] not in jellyfin_names]
print()
print("COMPTES AUTHENTIK SANS ÉQUIVALENT JELLYFIN")
print("-" * 70)
if not extras:
    print(" Aucun")
else:
    for u in extras:
        print(f" - {u['username']} [{u['type']}]")

print()
print("AUCUNE MODIFICATION EFFECTUÉE.")
print()
print("JSON :", out.relative_to(ROOT) / "cross-audit.json")
print("CSV  :", out.relative_to(ROOT) / "cross-audit.csv")
```

if **name** == "**main**": main()

migrate_one.py — cliquer pour afficher

```
#!/usr/bin/env python3
import argparse
import base64
import copy
import hashlib
import json
import os
import re
import sqlite3
import subprocess
import urllib.request
from pathlib import Path
```

ROOT = Path(**file**).resolve().parent AUTHENTIK_DIR = Path(os.environ.get("AUTHENTIK_DIR", "/mnt/user/appdata/compose/authentik")) DB_PATH = Path(os.environ.get("JF_DB", "/mnt/user/appdata/Jellyfin/data/jellyfin.db")) JF_URL = os.environ.get("JF_URL", "http://127.0.0.1:8096").rstrip("/") JF_API = os.environ.get("JF_API", "")

DEFAULT_PROVIDER = "Jellyfin.Server.Implementations.Users.DefaultAuthenticationProvider" LDAP_PROVIDER = os.environ.get("JF_LDAP_PROVIDER", "Jellyfin.Plugin.LDAP_Auth.LdapAuthenticationProviderPlugin",) PROTECTED = { value for value in os.environ.get("JF_PROTECTED_LOCAL_ADMINS", "admin").split(",") if value } TEST_ACCOUNTS = { value for value in os.environ.get("JF_TEST_ACCOUNTS", "hashtest").split(",") if value }

HASH_RE = re.compile(r"^$PBKDF2-SHA512$iterations=(\d+)$([0-9A-Fa-f]+)$([0-9A-Fa-f]+)$") RIGHTS_POLICY_EXCLUDE = { "AuthenticationProviderId", "PasswordResetProviderId", "InvalidLoginAttemptCount", }

def latest_dir(root): dirs = sorted(p for p in root.iterdir() if p.is_dir()) if root.exists() else [] if not dirs: raise RuntimeError(f"Aucun audit dans {root}") return dirs[-1]

def api_request(method, path, payload=None): if not JF_API: raise RuntimeError("JF_API n'est pas exportée.")

```
data = None
headers = {
    "Accept": "application/json",
    "Authorization": (
        'MediaBrowser '
```

```
        f'Token="{JF_API}", '
        'Client="jellyfin-authentik-migration", '
        'Device="Unraid", '
        'DeviceId="jellyfin-authentik-migration", '
        'Version="1.0"'
    ),
}
if payload is not None:
    data = json.dumps(payload).encode("utf-8")
    headers["Content-Type"] = "application/json"

req = urllib.request.Request(
    f"{JF_URL}{path}", data=data, headers=headers, method=method
)
with urllib.request.urlopen(req, timeout=30) as response:
    body = response.read()
    return None if not body else json.loads(body)
```

def canonical_hash(obj): raw = json.dumps(obj, ensure_ascii=False, sort_keys=True, separators=(",", ":")).encode("utf-8") return hashlib.sha256(raw).hexdigest()

def rights_fingerprint(user): policy = dict(user.get("Policy") or {}) for key in RIGHTS_POLICY_EXCLUDE: policy.pop(key, None) return canonical_hash({"Policy": policy, "Configuration": user.get("Configuration") or {}})

def jellyfin_hash_to_jf512(user_id): uri = f"file:{DB_PATH}?mode=ro" conn = sqlite3.connect(uri, uri=True) conn.row_factory = sqlite3.Row try: row = conn.execute("SELECT Password FROM Users WHERE Id = ?", (user_id,)).fetchone() finally: conn.close()

```
if row is None:
    raise RuntimeError("Utilisateur absent de la base Jellyfin.")

match = HASH_RE.match(row["Password"] or "")
if not match:
    raise RuntimeError("Hash Jellyfin non PBKDF2-SHA512.")

iterations = int(match.group(1))
salt_hex = match.group(2)
digest_hex = match.group(3)
```

```
if iterations != 210000:
    raise RuntimeError(
        f"Nombre d'itérations non validé : {iterations}; 210000 attendu par ce guide."
    )
if len(salt_hex) != 32:
    raise RuntimeError("Salt Jellyfin inattendu : 16 octets attendus.")
if len(digest_hex) != 128:
    raise RuntimeError("Digest Jellyfin inattendu : 64 octets attendus.")

salt = base64.urlsafe_b64encode(bytes.fromhex(salt_hex)).decode("ascii").rstrip("=")
digest = base64.urlsafe_b64encode(bytes.fromhex(digest_hex)).decode("ascii").rstrip("=")
return f"jf512${iterations}${salt}${digest}"
```

def authentik_helper(payload): proc = subprocess.run(["docker", "compose", "exec", "-T", "server", "ak", "shell", "-c", 'exec(open("/data/jellyfin_migration_helper.py").read())',], cwd=AUTHENTIK_DIR, input=json.dumps(payload, ensure_ascii=False) + "\n", text=True, capture_output=True,)

```
json_lines = [
    line.strip() for line in proc.stdout.splitlines() if line.strip().startswith("{")
]
if not json_lines:
    raise RuntimeError(
        "Aucun JSON retourné par le helper Authentik.\n"
        + proc.stdout[-2000:] + "\n" + proc.stderr[-2000:]
    )

result = json.loads(json_lines[-1])
if proc.returncode != 0 or not result.get("ok"):
    raise RuntimeError(
        "Helper Authentik en échec : " + json.dumps(result, ensure_ascii=False)
    )
return result
```

def load_reference(username): snap = latest_dir(ROOT / "snapshots") cross = latest_dir(ROOT / "cross-audits")

```
jf = json.loads((snap / "jellyfin-audit.json").read_text(encoding="utf-8"))
cross_data = json.loads((cross / "cross-audit.json").read_text(encoding="utf-8"))
```

```
    jf_matches = [u for u in jf["users"] if u.get("Name") == username]
    if len(jf_matches) != 1:
        raise RuntimeError("Utilisateur absent du snapshot Jellyfin.")

    cross_matches = [u for u in cross_data["users"] if u.get("username") == username]
    if len(cross_matches) != 1:
        raise RuntimeError("Utilisateur absent du cross-audit.")

    return jf_matches[0], cross_matches[0], snap, cross
```

def main(): parser = argparse.ArgumentParser() parser.add_argument("--user", required=True, help="Username Jellyfin exact") parser.add_argument("--apply", action="store_true", help="Effectuer réellement la migration") args = parser.parse_args()

```
    username = args.user

    try:
        snapshot_user, cross_user, snap, cross = load_reference(username)
    except Exception as exc:
        raise SystemExit(f"□ {exc}")

    if username in PROTECTED:
        raise SystemExit("□ Compte protégé : migration refusée.")
    if username in TEST_ACCOUNTS:
        raise SystemExit("□ Compte de test : migration production refusée.")
    if cross_user.get("is_disabled"):
        raise SystemExit("□ Compte désactivé : migration automatique refusée.")
    if cross_user.get("status") != "READY_NEW":
        raise SystemExit(f"□ Statut cross-audit incompatible : {cross_user.get('status')}")

    user_id = snapshot_user["Id"]
    original_policy = copy.deepcopy(snapshot_user.get("Policy") or {})
    original_configuration = copy.deepcopy(snapshot_user.get("Configuration") or {})
    original_rights = snapshot_user.get("RightsFingerprint")

    try:
        live = api_request("GET", f"/Users/{user_id}")
    except Exception as exc:
        raise SystemExit(f"□ Impossible de lire Jellyfin live : {exc}")
```

```
if live.get("Name") != username:
    raise SystemExit(
        "⚠ Drift : username live différent du snapshot : "
        f"{live.get('Name')!r} != {username!r}"
    )
if live.get("Id") != user_id:
    raise SystemExit("⚠ Drift : Jellyfin ID différent.")
if (live.get("Policy") or {}) != original_policy:
    raise SystemExit("⚠ Drift : Policy Jellyfin modifiée depuis le snapshot.")
if (live.get("Configuration") or {}) != original_configuration:
    raise SystemExit("⚠ Drift : Configuration Jellyfin modifiée depuis le snapshot.")

try:
    encoded_hash = jellyfin_hash_to_jf512(user_id)
    auth = authentik_helper({"action": "inspect", "username": username})
except Exception as exc:
    raise SystemExit(f"⚠ Précontrôle impossible : {exc}")

if auth.get("exists"):
    raise SystemExit("⚠ Utilisateur Authentik déjà présent.")
case_matches = auth.get("case_matches") or []
if case_matches:
    raise SystemExit(f"⚠ Collision de casse Authentik : {case_matches!r}")

print()
print("=" * 70)
print(" MIGRATION JELLYFIN → AUTHENTIK")
print("=" * 70)
print()
print("Utilisateur              :", username)
print("Jellyfin ID              :", user_id)
print("Admin                     :", bool(original_policy.get("IsAdministrator")))
print("Désactivé                 :", bool(original_policy.get("IsDisabled")))
print("Provider actuel           :", original_policy.get("AuthenticationProviderId"))
print("Provider cible            :", LDAP_PROVIDER)
print("PasswordResetProvider     :", original_policy.get("PasswordResetProviderId"))
print()
print("Hash Jellyfin             : PBKDF2-SHA512 compatible")
```

```
print("Hash Authentik transitoire: jf512")
print("Utilisateur Authentik     : absent")
print()
print("RightsFingerprint         :", original_rights)
print()
print("PasswordResetProviderId NE SERA PAS MODIFIÉ.")

if not args.apply:
    print()
    print("□□DRY-RUN : AUCUNE MODIFICATION EFFECTUÉE.")
    print()
    print("Pour migrer réellement cet utilisateur :")
    print(f"./migrate_one.py --user {username!r} --apply")
    return

print()
print("⚠□ MODE APPLY")
auth_created = False
jellyfin_changed = False

try:
    print()
    print("[1/4] Création Authentik...")
    created = authentik_helper(
        {
            "action": "create",
            "username": username,
            "encoded_hash": encoded_hash,
            "jellyfin_id": user_id,
        }
    )
    auth_created = True
    if created.get("algorithm") != "jf512":
        raise RuntimeError("Authentik n'a pas stocké jf512.")
    if "jellyfin_users" not in created.get("groups", []):
        raise RuntimeError("Groupe jellyfin_users absent après création.")
    print("      □ utilisateur créé")
    print("      □ hash jf512 importé")
    print("      □ groupe jellyfin_users")
```

```
print()
print("[2/4] Vérification Authentik...")
check = authentik_helper({"action": "inspect", "username": username})
if not check.get("exists"):
    raise RuntimeError("Utilisateur Authentik absent après création.")
if check.get("algorithm") != "jf512":
    raise RuntimeError("Algorithm Authentik inattendu.")
if "jellyfin_users" not in check.get("groups", []):
    raise RuntimeError("jellyfin_users absent.")
print("      utilisateur présent")
print("      algorithm = jf512")
print("      jellyfin_users présent")

print()
print("[3/4] Changement AuthenticationProviderId...")
new_policy = copy.deepcopy(original_policy)
new_policy["AuthenticationProviderId"] = LDAP_PROVIDER
# Send the FULL policy. Provider/reset-provider fields are required in
# current Jellyfin DB paths and must not be accidentally nulled.
api_request("POST", f"/Users/{user_id}/Policy", new_policy)
jellyfin_changed = True

print()
print("[4/4] Contrôle intégral après migration...")
after = api_request("GET", f"/Users/{user_id}")
after_policy = after.get("Policy") or {}
after_configuration = after.get("Configuration") or {}

expected_policy = copy.deepcopy(original_policy)
expected_policy["AuthenticationProviderId"] = LDAP_PROVIDER

if after_policy != expected_policy:
    changed = sorted(
        key
        for key in set(original_policy) | set(after_policy)
        if original_policy.get(key) != after_policy.get(key)
    )
    raise RuntimeError(
```

```
            "Policy finale différente de l'attendu. Champs différents : "
            + ", ".join(changed)
        )
    if after_configuration != original_configuration:
        raise RuntimeError("Configuration Jellyfin modifiée.")
    if rights_fingerprint(after) != original_rights:
        raise RuntimeError("RightsFingerprint différent après migration.")

    print("      seul AuthenticationProviderId a changé")
    print(f"      {len(original_policy)} champs Policy protégés")
    print(f"      {len(original_configuration)} champs Configuration protégés")
    print("      RightsFingerprint identique")

except Exception as exc:
    print()
    print("ERREUR DE MIGRATION :", exc)
    print("ROLLBACK AUTOMATIQUE...")
    rollback_errors = []

    if jellyfin_changed:
        try:
            api_request("POST", f"/Users/{user_id}/Policy", original_policy)
            print("      Policy Jellyfin restaurée")
        except Exception as rb_exc:
            rollback_errors.append(f"Jellyfin: {rb_exc}")

    if auth_created:
        try:
            authentik_helper(
                {"action": "rollback", "username": username, "jellyfin_id": user_id}
            )
            print("      utilisateur Authentik de migration supprimé")
        except Exception as rb_exc:
            rollback_errors.append(f"Authentik: {rb_exc}")

    if rollback_errors:
        print("ROLLBACK INCOMPLET :", "; ".join(rollback_errors))
    raise SystemExit(1)
```

```
print()
print("=" * 70)
print("  MIGRATION TECHNIQUE RÉUSSIE")
print("=" * 70)
print()
print("Utilisateur :", username)
print("Jellyfin    : LDAP")
print("Authentik   : jf512")
print()
print("Il reste maintenant à TESTER UNE CONNEXION avec le mot de passe Jellyfin
historique.")
print()
print("Après cette première connexion, Authentik doit convertir jf512 → pbkdf2_sha256.")
```

if **name** == "**main**": main()

make_batch.py — cliquer pour afficher

```
#!/usr/bin/env python3
import argparse
import json
from pathlib import Path
```

ROOT = Path(**file**).resolve().parent

def latest_cross(): root = ROOT / "cross-audits" dirs = sorted(p for p in root.iterdir() if p.is_dir()) if root.exists() else [] if not dirs: raise SystemExit(" Aucun cross-audit.") return dirs[-1]

def main(): parser = argparse.ArgumentParser() parser.add_argument("--count", type=int, required=True) parser.add_argument("--output", required=True) args = parser.parse_args()

```
cross_dir = latest_cross()
data = json.loads((cross_dir / "cross-audit.json").read_text(encoding="utf-8"))
ready = sorted(
    (u["username"] for u in data["users"] if u["status"] == "READY_NEW"),
    key=str.casefold,
)

if len(ready) &lt; args.count:
```

```
    raise SystemExit(
        f"⚠ Seulement {len(ready)} READY_NEW pour {args.count} demandés."
    )

selected = ready[: args.count]
output = ROOT / args.output
output.write_text("\n".join(selected) + "\n", encoding="utf-8")

print("Cross-audit :", cross_dir.relative_to(ROOT))
print("READY_NEW total :", len(ready))
print("Batch créé :", output.relative_to(ROOT))
print()
for i, username in enumerate(selected, 1):
    print(f"{i:03d}. {username!r}")
```

if **name** == "**main**": main()

run_dry_batch.py — cliquer pour afficher

```
#!/usr/bin/env python3
import argparse
import datetime as dt
import subprocess
import sys
from pathlib import Path
```

ROOT = Path(**file**).resolve().parent

def main(): parser = argparse.ArgumentParser() parser.add_argument("--file", required=True) parser.add_argument("--count", type=int, required=True) args = parser.parse_args()

```
batch = Path(args.file)
if not batch.is_absolute():
    batch = ROOT / batch
users = [line for line in batch.read_text(encoding="utf-8").splitlines() if line != ""]

if len(users) != args.count:
    raise SystemExit(f"⚠ {args.count} attendus, {len(users)} trouvés")
```

```
log_dir = ROOT / "dry-run-logs"
log_dir.mkdir(exist_ok=True)
log = log_dir / f"{batch.stem}-{dt.datetime.now().strftime('%Y%m%d-%H%M%S')}.log"

success = 0
with log.open("w", encoding="utf-8", buffering=1) as fh:
    for i, username in enumerate(users, 1):
        banner = f"\n{'='*70}\nDRY-RUN [{i}/{len(users)}] {username!r}\n{'='*70}\n"
        print(banner, end="", flush=True)
        fh.write(banner)
        fh.flush()

        proc = subprocess.Popen(
            [sys.executable, str(ROOT / "migrate_one.py"), "--user", username],
            cwd=ROOT,
            stdout=subprocess.PIPE,
            stderr=subprocess.STDOUT,
            text=True,
            bufsize=1,
        )
        assert proc.stdout is not None
        for line in proc.stdout:
            print(line, end="", flush=True)
            fh.write(line)
            fh.flush()
        rc = proc.wait()
        if rc != 0:
            print(f"□ Dry-run échoué pour {username!r}. Arrêt.")
            print("Log :", log.relative_to(ROOT))
            raise SystemExit(1)
        success += 1

print()
print(f"□ Dry-runs réussis : {success}/{len(users)}")
print("Log :", log.relative_to(ROOT))
```

if **name** == "**main**": main()

run_batch_safe.py — cliquer pour afficher

```
#!/usr/bin/env python3
import argparse
import datetime as dt
import re
import subprocess
import sys
from pathlib import Path
```

ROOT = Path(**file**).resolve().parent

def safe_filename(value): value = re.sub(r"[^A-Za-z0-9.-]+", "", value) return value[:80] or "user"

def main(): parser = argparse.ArgumentParser() parser.add_argument("--file", required=True) parser.add_argument("--count", required=True, type=int) args = parser.parse_args()

```
batch_file = Path(args.file)
if not batch_file.is_absolute():
    batch_file = ROOT / batch_file
if not batch_file.exists():
    raise SystemExit(f"❌ Batch absent : {batch_file}")

raw_lines = batch_file.read_text(encoding="utf-8").splitlines()

# CRITICAL: do not strip usernames. A real migration encountered a Jellyfin
# username with a trailing space. Removing it would silently change identity.
users = [line for line in raw_lines if line != ""]

if len(users) != args.count:
    raise SystemExit(
        f"❌ Sécurité : {args.count} comptes attendus, {len(users)} trouvés."
    )
if len(set(users)) != len(users):
    raise SystemExit("❌ Sécurité : usernames dupliqués dans le batch.")

print("BATCH JELLYFIN → AUTHENTIK")
print("=" * 70)
```

```
print()
print("Fichier :", batch_file)
print("Comptes :", len(users))
print()
for i, username in enumerate(users, 1):
    print(f"{i:03d}. {username!r}")

print()
confirmation = input(f"Tapez exactement MIGRATE {len(users)} pour continuer : ")
if confirmation != f"MIGRATE {len(users)}":
    raise SystemExit("Annulé.")

stamp = dt.datetime.now().strftime("%Y%m%d-%H%M%S")
log_dir = ROOT / "batch-logs" / stamp
log_dir.mkdir(parents=True, exist_ok=False)

results = []

for index, username in enumerate(users, 1):
    print()
    print("=" * 70)
    print(f"[{index}/{len(users)}] {username!r}")
    print("=" * 70)

    log_path = log_dir / f"{index:03d}-{safe_filename(username)}.log"
    proc = subprocess.Popen(
        [
            sys.executable,
            str(ROOT / "migrate_one.py"),
            "--user",
            username,
            "--apply",
        ],
        cwd=ROOT,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True,
        bufsize=1,
```

```
    )

    with log_path.open("w", encoding="utf-8", buffering=1) as log:
        assert proc.stdout is not None
        for line in proc.stdout:
            print(line, end="", flush=True)
            log.write(line)
            log.flush()

    rc = proc.wait()
    if rc == 0:
        results.append((username, "MIGRATED", rc))
        print(f"□ {username!r}: MIGRATED")
    else:
        results.append((username, "FAILED", rc))
        print(f"□ {username!r}: FAILED")
        break

summary_path = log_dir / "summary.txt"
with summary_path.open("w", encoding="utf-8") as fh:
    fh.write("BATCH JELLYFIN → AUTHENTIK\n")
    fh.write("=" * 70 + "\n\n")
    for username, state, rc in results:
        fh.write(f"{username!r}: {state} (rc={rc})\n")

failed = [r for r in results if r[1] == "FAILED"]
print()
if failed:
    print("⚠□ Batch incomplet.")
    print("Ne pas poursuivre avant audit.")
elif len(results) == len(users):
    print("□ Batch terminé avec succès.")
else:
    print("⚠□ Batch interrompu avant la fin.")
print()
print("Résumé :", summary_path.relative_to(ROOT))

if failed:
```

```
    raise SystemExit(1)
```

if **name** == "**main**": main()

35. Sources officielles utiles

- Authentik — Docker Compose installation
- Authentik — Create an LDAP provider
- Authentik — LDAP Provider / TLS
- Authentik — Manual Outpost deployment in Docker Compose
- Authentik 2025.10 — suppression de Redis
- Authentik 2026.5 release notes
- Authentik source — chargement de data.user_settings
- Authentik source — User.set_password_from_hash / check_password
- Jellyfin — Plugins / LDAP Authentication
- Jellyfin — plugin LDAP officiel
- Jellyfin LDAP — comportement Admin Filter dans le code source

36. Conclusion

Une migration Jellyfin vers Authentik peut être réalisée sans recréer les comptes et sans imposer immédiatement un nouveau mot de passe, à condition de traiter séparément quatre sujets : **l’identité, les droits Jellyfin, le hash de mot de passe et la récupération de mot de passe**.

Les garde-fous qui ont réellement fait la différence pendant la migration sont : **un admin local de secours, des snapshots en lecture seule, un cross-audit, un dry-run par compte, un batch séquentiel qui s’arrête au premier échec, un rollback marqué par Jellyfin ID, le POST de la Policy complète, le LDAP Admin Filter désactivé, la conservation exacte des usernames et le maintien du hasher jf512 jusqu’à la dernière première connexion**.

Le résultat est une base saine pour faire d’Authentik le point central d’identité de l’écosystème self-hosted, sans sacrifier la compatibilité native de Jellyfin.

Revision #4
Created 2026-08-15 20:17:40 UTC by thymon
Updated 2026-08-16 01:19:41 UTC by thymon