RAPPORT TECHNIQUE DE DÉVELOPPEMENT

Application Web de Synchronisation DHIS2

E-Santé

(09/10/2025)

Table des matières

OBJECT PRINCIPAL

Synchroniser automatiquement toutes les métadonnées et les données d'une instance DHIS2 source vers une instance DHIS2 cible. Le but c'est de garantir un référentiel commun et des chiffres comparables partout, tout en acheminant les données opérationnelles au bon endroit, au bon moment.

Son objectif principal est de faciliter et automatiser la réplication de données et de métadonnées entre différentes instances DHIS2, permettant ainsi l'interopérabilité et la consolidation d'informations sanitaires provenant de multiples sources.

BUTS DE L'APPLICATION

L'application vise à répondre aux besoins suivants :

Synchronisation Bidirectionnelle et Multi-Instance

- Permettre la synchronisation entre plusieurs instances DHIS2 (source et destination)
- Gérer plusieurs configurations de synchronisation simultanément
- Supporter différents modes d'exécution : manuel, automatique et planifié

Préservation de l'Intégrité des Données

- Respecter les dépendances entre les différents types de métadonnées lors de la synchronisation
- Garantir l'ordre correct d'importation (métadonnées avant données, catégories avant éléments de données, etc.)
- Valider la compatibilité entre les versions DHIS2 source et destination

Automatisation et Monitoring

- Détection automatique des changements sur l'instance source
- Déclenchement automatique des synchronisations en fonction des modifications détectées
- Surveillance continue avec intervalles configurables

Flexibilité et Granularité

- Synchronisation sélective par type de données (métadonnées, données agrégées, événements, tracker)
- Synchronisation par famille de métadonnées (organisation, utilisateurs, programmes, etc.)
- Filtrage par période, unités organisationnelles, ou programmes spécifiques

Traçabilité et Suivi

- Historique complet des synchronisations avec logs détaillés

- Système de jobs avec statuts (en attente, en cours, terminé, échoué)
- Gestion automatique des erreurs avec mécanisme de retry

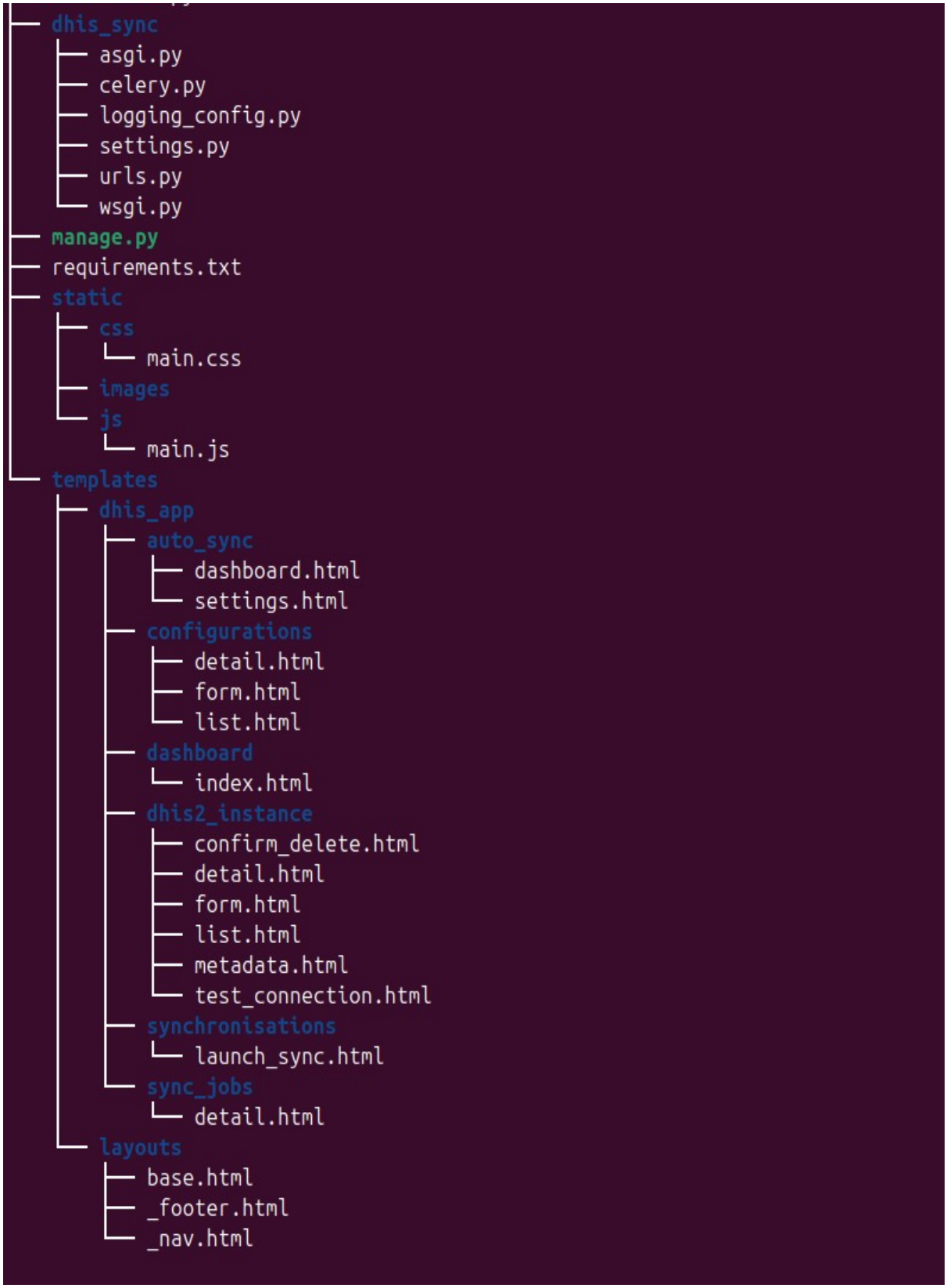
ARCHITECTURE ET STRUCTURE TECHNIQUE

Technologies utilisées

Couche	Technologie	Description
Frontend (Interface Utilisateur)	**Bootstrap 5, HTML5, CSS3, JavaScript**	Permet une interface web moderne, responsive et simple d'utilisation.
Backend (Serveur Applicatif)	**Django 4.2 (Python)**	Framework robuste et sécurisé pour la gestion des APIs, de la logique métier et de la persistance des données.
Base de données	**PostgreSQL**	Système de gestion de base de données relationnelle fiable et performant.
Tâches asynchrones	**Celery + Redis**	Gestion des synchronisations automatiques et planifiées sans bloquer le serveur principal.
API de communication	**dhis2.py**	Bibliothèque Python utilisée pour interagir avec les API REST des instances DHIS2.

Structure de l'Application

Structure du code de l'Application


dhis_sync
asgi.py
celery.py
logging_config.py
settings.py
urls.py
wsgi.py
manage.py
requirements.txt
static
css
main.css
images
js
main.js
templates
dhis_app
auto_sync
dashboard.html
settings.html
configurations
detail.html
form.html
list.html
dashboard
index.html
dhis2_instance
confirm_delete.html
detail.html
form.html
list.html
metadata.html
test_connection.html
synchronisations
launch_sync.html
sync_jobs
detail.html
layouts
base.html
_footer.html
_nav.html

FONCTIONNALITÉS DÉTAILLÉES

Configuration des Instances

Configuration des instances (dhis_app/models.py)

- Enregistrement de multiples instances DHIS2 avec leurs paramètres de connexion
- Test automatique de connectivité via l'API DHIS2
- Détection de la version DHIS2 de chaque instance
- Vérification périodique du statut de connexion

Compatibilité inter-versions

- Mapping automatique des champs entre versions DHIS2 différentes
- Détection des champs dépréciés et nouveaux
- Transformations de données adaptatives selon les versions

Configuration de la synchronisation

Types de synchronisation multiples (dhis_app/models.py)

- Métadonnées uniquement : Structure et configuration du système
- Données agrégées : Valeurs de données collectées via formulaires de saisie
- Événements : Données événementielles des programmes
- Tracker : Entités suivies (patients, bénéficiaires) avec leurs inscriptions
- Synchronisation complète : Orchestration de tous les types avec gestion des dépendances

Modes d'exécution

- Manuel : Déclenchement à la demande par l'utilisateur
- Automatique : Surveillance continue avec détection de changements
- Planifié : Exécution selon un calendrier prédéfini

Synchronisation

Une synchronisation consiste à exporter des données ou métadonnées d'une instance source et les importer une instances cible ou de destination de DHIS2. Cela peut s'effectuer d'une manière automatique ou manuelle en créant un objet python SynJob à partir duquel on peut :
Lancer une synchronisation soit des données (voir même des données spécifiques) ou des métadonnées uniquement, soit les deux dans un seul processus (en suivant un ordre précis).
La synchronisation des données prend en compte tous les types de données (agrégées, d'évènements et tracker).
La synchronisation des métadonnées prend en compte toutes les familles

Organisation en familles (dhis_app/services/metadata/metadata_sync_service.py)

L'application organise les métadonnées en 14 familles logiques :

- Users : Utilisateurs, rôles, groupes d'utilisateurs
- Organisation : Niveaux, unités organisationnelles, groupes d'unités
- Categories : Options de catégories, catégories, combinaisons
- Options : Ensembles d'options, options
- Data Elements : Éléments de données, groupes, opérandes
- Indicators : Types d'indicateurs, indicateurs, groupes
- Data Sets : Ensembles de données et leurs éléments
- Tracker : Types d'entités suivies, attributs
- Programs : Programmes, étapes, règles de programmes
- Validation : Règles de validation, groupes
- Predictors : Prédicteurs et leurs groupes
- Legends : Légendes et ensembles de légendes
- System : Paramètres système, attributs, constantes
- Analytics : Visualisations, cartes, tableaux, dashboards

Orchestration de Synchronisation (dhis_app/services/sync_orchestrator.py)

Gestion de l'ordre d'exécution

- Métadonnées en premier (prérequis pour les données)
- Données Tracker, puis Événements, puis Agrégées
- Vérification de compatibilité avant démarrage
- Jobs composites avec sous-jobs pour chaque étape

Suivi de progression en temps réel

- Progression en pourcentage pour chaque job
- Compteurs détaillés : total, traités, succès, erreurs
- Logs structurés avec horodatage
- Statuts multiples : pending, running, completed, completed_with_warnings, failed

Gestion des erreurs et retry

- Retry automatique avec backoff exponentiel
- Maximum de 3 tentatives par job
- Cooldown configurable après erreur
- Jobs enfants pour les retries

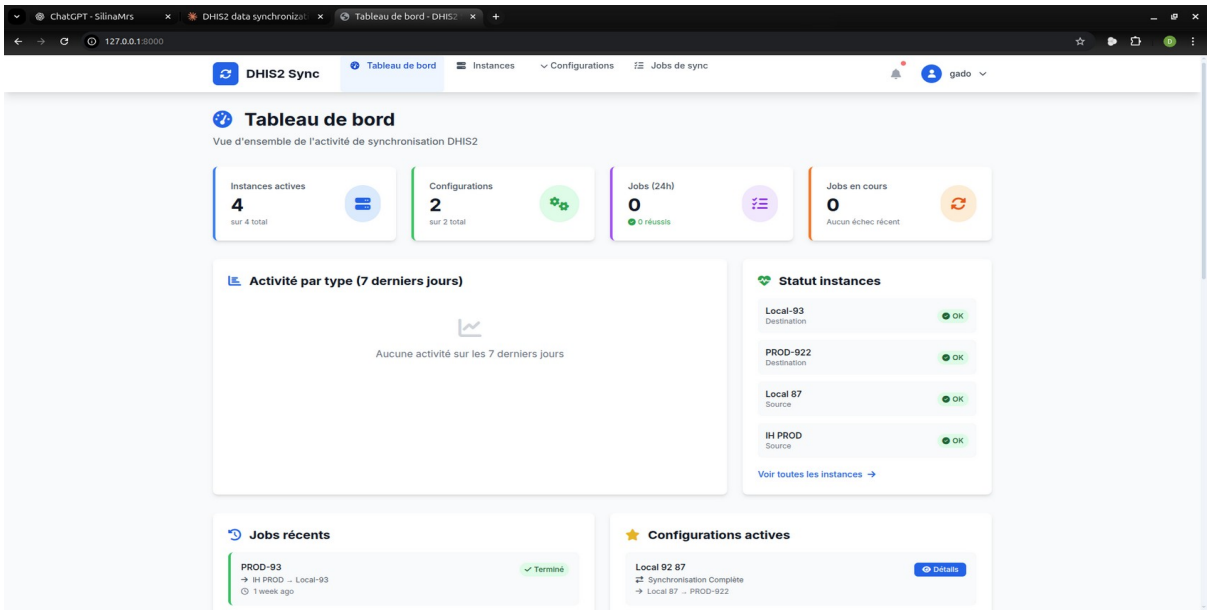
Synchronisation Automatique (dhis_app/services/auto_sync/)

- Détection de changements (change_detector.py)
- Surveillance périodique via l'API DHIS2 avec filtre lastUpdated

INTERFACE UTILISATEUR (CLIENT)

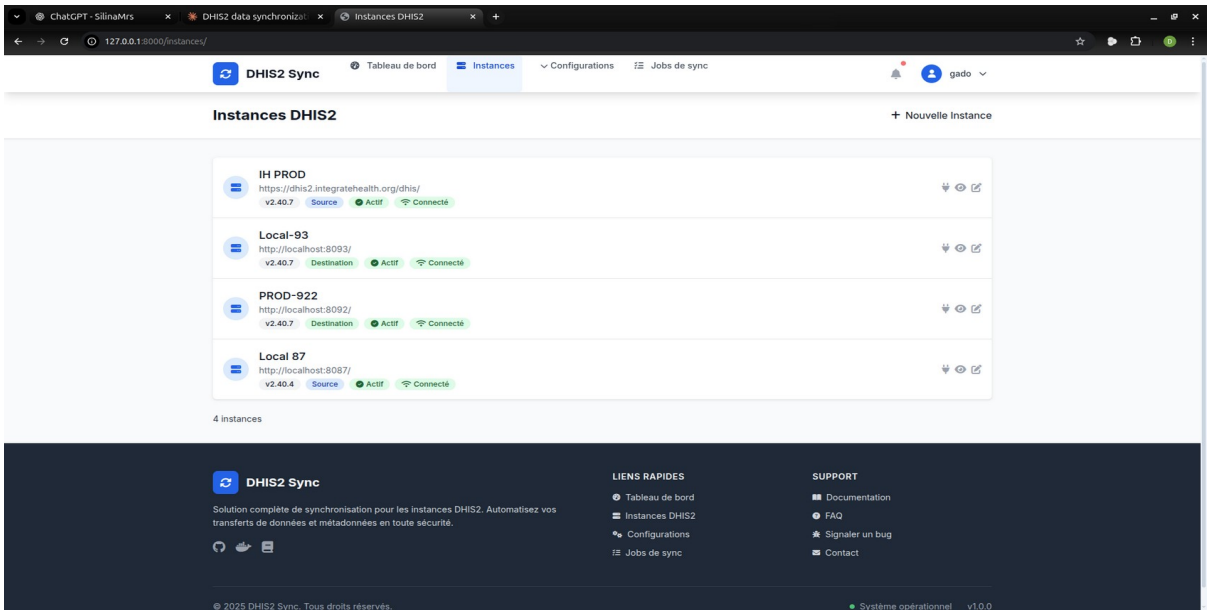
- Vue d'ensemble : Statistiques globales des synchronisations
- Jobs récents : derniers jobs avec statuts

- Instances actives : État de connexion des instances DHIS2

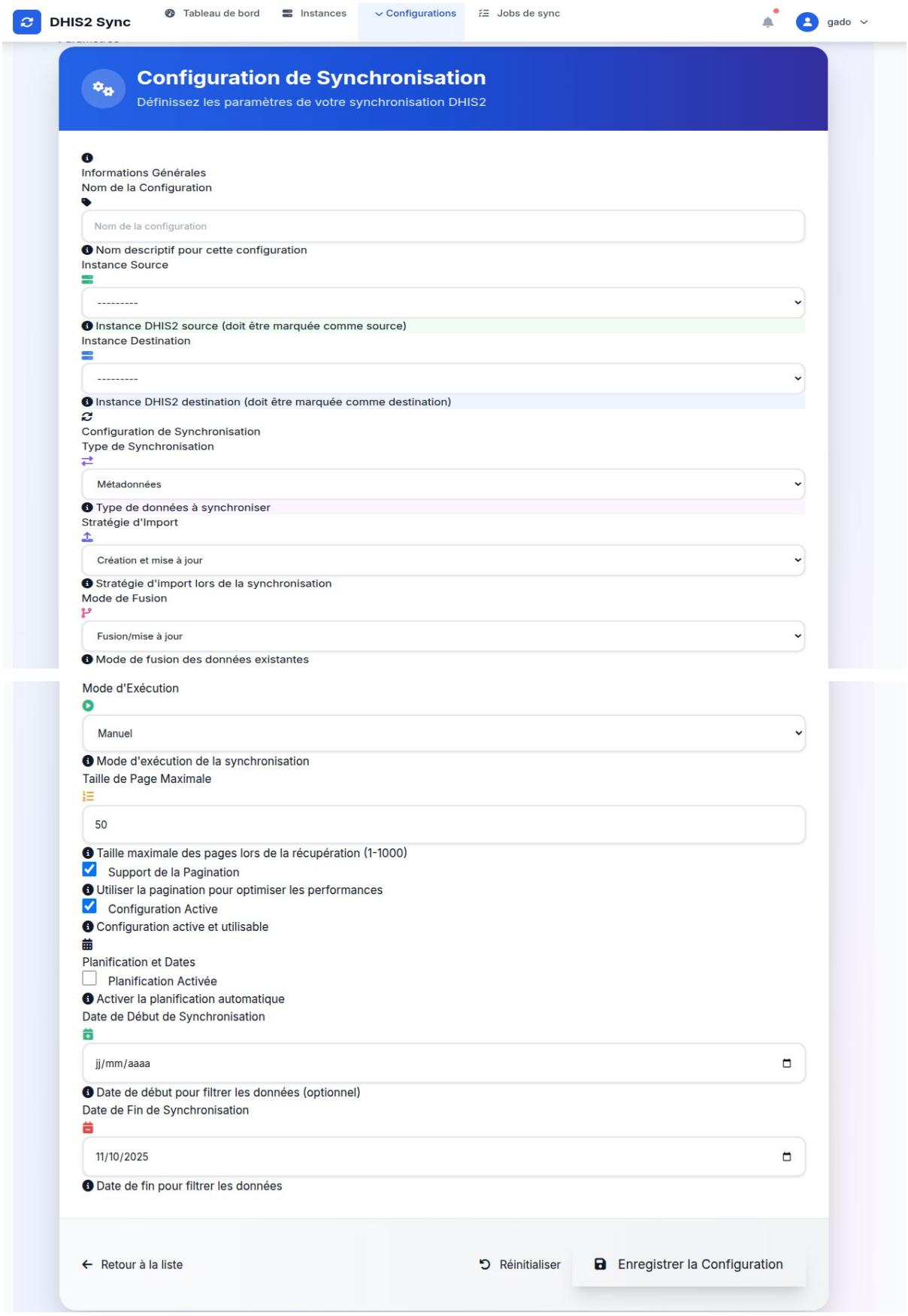


Gestion des Instances DHIS2

- Création/Modification : Formulaire de configuration instance
- Test de connexion : Vérification automatique de la connectivité
- Types d'instance : Source et Destination
- Statut temps réel : Indicateur visuel de l'état de connexion



Configuration de la synchronisation des instances

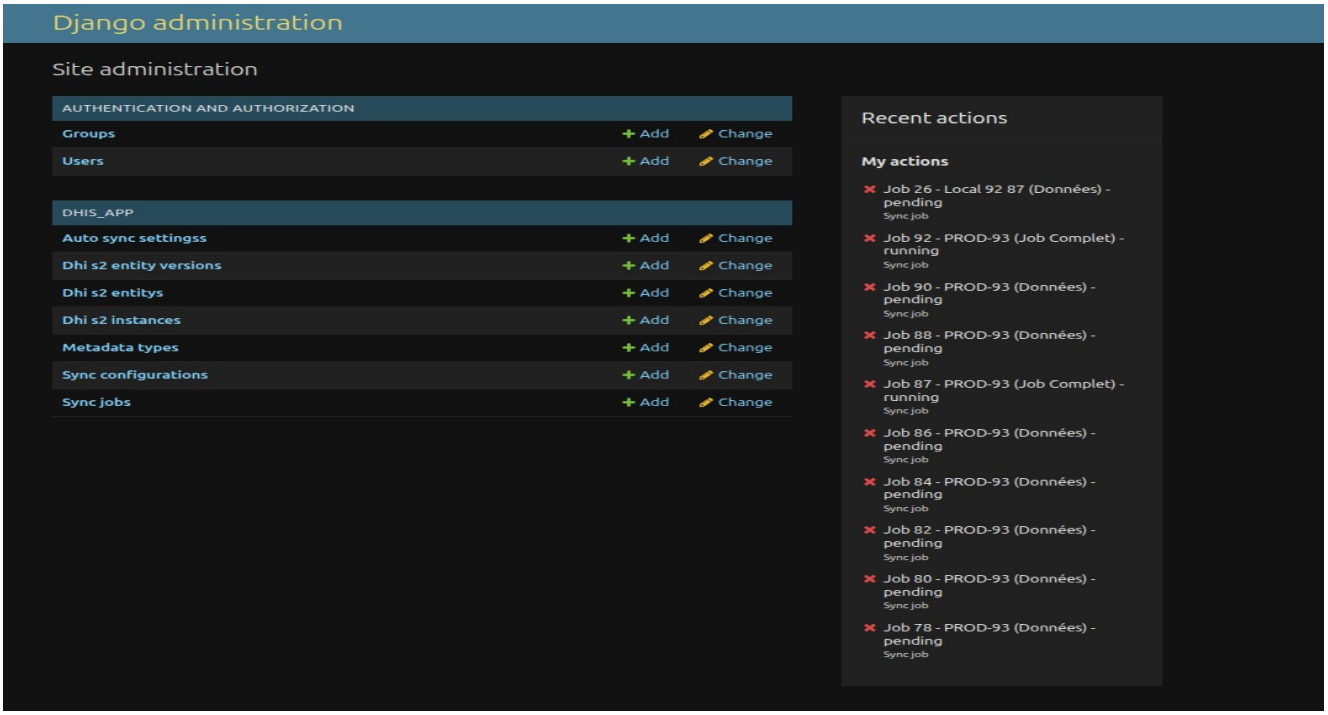

DHIS2 Sync
Tableau de bord
Instances
Configurations
Jobs de sync
gado
Configuration de Synchronisation
Définissez les paramètres de votre synchronisation DHIS2
Informations Générales
Nom de la Configuration
Nom de la configuration
Nom descriptif pour cette configuration
Instance Source

Instance DHIS2 source (doit être marquée comme source)
Instance Destination

Instance DHIS2 destination (doit être marquée comme destination)
Configuration de Synchronisation
Type de Synchronisation
Métadonnées
Type de données à synchroniser
Stratégie d'Import
Création et mise à jour
Stratégie d'import lors de la synchronisation
Mode de Fusion
Fusion/mise à jour
Mode de fusion des données existantes
Mode d'Exécution
Manuel
Mode d'exécution de la synchronisation
Taille de Page Maximale
50
Taille maximale des pages lors de la récupération (1-1000)
Support de la Pagination
Utiliser la pagination pour optimiser les performances
Configuration Active
Configuration active et utilisable
Planification et Dates
Planification Activée
Activer la planification automatique
Date de Début de Synchronisation
jj/mm/aaaa
Date de début pour filtrer les données (optionnel)
Date de Fin de Synchronisation
11/10/2025
Date de fin pour filtrer les données
Retour à la liste
Réinitialiser
Enregistrer la Configuration

Interface Administration

Cette interface d’administration est accessible uniquement par les administrateurs et ceux ayant les droits d’accès



CONCLUSION

L’application web de synchronisation DHIS2 constitue une **solution robuste, automatisée et extensible** répondant aux besoins de synchronisation des données et métadonnées entre deux instances (source et destination).
Elle permet ainsi la réplication des données d’une instance de DHIS2 .

ANNEXE

Code de la vue qui lance la synchronisation des métadonnées

```python
class LaunchMetadataSyncView(LaunchSynchronizationView):
    """Vue pour lancer uniquement la synchronisation des métadonnées"""

    def post(self, request, config_id):
        config = self.get_config(config_id)

        # Validation
        error_message = self.validate_config(config)
        if error_message:
            return self.handle_error(request, error_message, config.id)

        try:
            # Paramètres
            families = request.POST.getlist('metadata_families') or None

            # Créer l'orchestrateur et lancer la synchronisation
            orchestrator = SyncOrchestrator(config)

            job = orchestrator.execute_metadata_sync(
                sync_config=config,
                families=families
            )

            success_message = f'Synchronisation des métadonnées lancée (Job #{job.id})'
            return self.handle_success(request, success_message, job.id)

        except Exception as e:
            logger.error(f"Erreur lors du lancement de la synchronisation métadonnées: {e}")
            error_message = f'Erreur lors du lancement: {str(e)}'
            return self.handle_error(request, error_message, config.id)
```

Code de la vue qui lance la synchronisation des données

```python
class LaunchDataSyncView(LaunchSynchronizationView):
    """Vue pour lancer uniquement la synchronisation des données"""

    def post(self, request, config_id):
        config = self.get_config(config_id)
```

```
    print("Je suis dans LaunchDataSyncView")
    # Validation
    error_message = self.validate_config(config)
    if error_message:
        return self.handle_error(request, error_message, config.id)

    try:
        # Paramètres
        #sync_types = request.POST.getlist('data_types')
        sync_types = ['events']
        if not sync_types:
            sync_types = ['tracker', 'events', 'aggregate']  # Par défaut tous

        org_units = request.POST.getlist('org_units') or None
        programs = request.POST.getlist('programs') or None
        periods = request.POST.getlist('periods') or None

        # Créer l'orchestrateur et lancer la synchronisation
        orchestrator = SyncOrchestrator(config)

        job = orchestrator.execute_data_sync(
            sync_config=config,
            sync_types=sync_types,
            org_units=org_units,
            programs=programs,
            periods=periods
        )

        success_message = f'Synchronisation des données lancée (Job #{job.id})'
        return self.handle_success(request, success_message, job.id)

    except Exception as e:
        logger.error(f"Erreur lors du lancement de la synchronisation données: {e}")
        error_message = f'Erreur lors du lancement: {str(e)}'
        return self.handle_error(request, error_message, config.id)

     }
```

Code de la methode de test de connexion à l'instance DHIS2

```
def test_connection(self):
    """
    Test la connexion à l'instance DHIS2
```

```
    Returns:
        dict: Résultat du test avec statut et informations
    """
    try:
        api = self.get_api_client()

        # Test basique avec l'endpoint system/info
        response = api.get('system/info')

        if response.status_code == 200:
            info = response.json()
            return {
                'success': True,
                'message': 'Connexion réussie',
                'dhis2_version': info.get('version'),
                'system_name': info.get('systemName'),
                'server_date': info.get('serverDate')
            }
        else:
            return {
                'success': False,
                'message': f'Erreur HTTP {response.status_code}: {response.text}'
            }

    except ImportError as e:
        return {
            'success': False,
            'message': str(e)
        }
    except ValidationError as e:
        return {
            'success': False,
            'message': str(e)
        }
    except Exception as e:
        return {
            'success': False,
            'message': f'Erreur de connexion: {str(e)}'
        }
```

Extrait du code des modèles

Modèle d’instance

```python
class DHIS2Instance(models.Model):
    name = models.CharField(max_length=100, unique=True)
    base_url = models.URLField()
    username = models.CharField(max_length=100)
    password = models.CharField(max_length=255)
    version = models.CharField(max_length=20, blank=True, null=True, help_text="Version DHIS2 (ex: 2.38, 2.40)")
    is_source = models.BooleanField(default=False)
    is_destination = models.BooleanField(default=False)
    is_active = models.BooleanField(default=True, help_text="Instance active/inactive")
    connection_status = models.BooleanField(default=None, null=True, blank=True, help_text="Statut de connexion (True=connecté, False=déconnecté, None=non testé)")
    last_connection_test = models.DateTimeField(null=True, blank=True, help_text="Dernière vérification de connexion")
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.name} ({'Source' if self.is_source else 'Destination'})"

    def clean(self):
        """Validation personnalisée"""
        super().clean()

        # Nettoyer l'URL pour éviter les double slashes
        if self.base_url:
            # Supprimer les slashes multiples à la fin
            self.base_url = self.base_url.rstrip('/')
            # S'assurer qu'il n'y a qu'un seul slash à la fin
            if not self.base_url.endswith('/'):
                self.base_url += '/'

        # Vérifier qu'au moins l'un des deux flags est activé
        if not self.is_source and not self.is_destination:
            raise ValidationError("Une instance doit être soit source, soit destination, soit les deux.")

    def get_api_client(self):
        """
        Crée et retourne un client API dhis2.py pour cette instance
        """
```

```
    try:
        # Nettoyer l'URL pour éviter les doubles slashes
        clean_url = self.base_url.rstrip('/') if self.base_url else ''

        api = Api(
            server = clean_url,  # Passer l'URL sans slash final
            username = self.username,
            password = self.password
        )

        return api

    except Exception as e:
        raise ValidationError(f"Impossible de créer le client API: {str(e)}")
}
```