

- Introduction
  - Préalable
    - exemple
    - Vocabulaire
      - Clefs et code
        - code
        - Clef naturelle.
        - Clef hiérarchique
      - Référentiels
      - Types de données
- Aide fichier à la rédaction du fichier de configuration
  - La création :
    - Description du fichier
      - Version de l'analyseur parser du fichier de configuration.
      - on présente l'application avec son nom et la version du fichier de configuration :
    - Description référentiels
      - Description des colonnes columns
      - Colonnes calculées computed columns
      - Colonnes dynamiques dynamic columns
      - On peut poser des contraintes sur les données de référence
        - Utilisation de vérificateurs checker
        - Utilisation de validations portant sur une ou plusieurs colonnes
        - Vérificateurs
    - <a id="compositeReferences" />Définition de clefs composites entre différentes références
      - Relation entre deux référentiels avec multiplicité
    - <a id="datatypes" />Description des *dataTypes*
      - *data*
      - la validation est utilisé pour valider une ligne sur une ou plusieurs colonnes.
      - Déclaration des contraintes d'unicité
      - ensuite on va décrire le format des données attendues dans *format* décrite dans la partie *dataTypes*:
        - Définition de constantes
        - Lien avec les colonnes
        - Lien avec les colonnes répétées
      - *authorization* Dans la section **authorization**, on définit les objets sur lesquels porteront les les autorisations d'accès aux données :
        - Groupe de variables datagroups
        - Portée des données authorizationScope.
        - Temporalité des données timeScope.
  - lors de l'importation du fichier yaml :
  - Internationalisation du fichier yaml:
    - Internationalisation de l'application:
    - Internationalisation des *references*:
    - Internationalisation des *dataTypes*:
  - Zip de YAML
  - lors de l'importation du fichier yaml :
- Aide fichier .csv
  - lors de l'ouverture du fichier csv via libre office:
  - lors de la création du fichier csv de Référence et de donnée :
  - lors de l'importation de fichier csv dans l'application:

# Introduction

---

Ce document permet d'aider un gestionnaire de SI à décrire son domaine dans un fichier de configuration, qui une fois déposé dans l'application, génèrera une base de données et les outils permettant de l'alimenter et de la consulter.

Chaque fichier de configuration déposé génèrera une schéma dédié dans la base de données.

## Préalable

Avant de débiter l'écriture du fichier de configuration, il faut travailler à définir le modèle des données que vous voulez traiter dans la base de données.

Vous avez en votre possession un certain nombre de fichiers (format csv) contenant les données. Un fichier de données respecte un certain format. En particulier les en-têtes de colonnes doivent être fixés, et le contenu sous un en-tête a un format déterminé (date, valeur flottante, entier, texte..).

Chaque format de fichier correspond à ce que l'on appellera un type de données. Il regroupe plusieurs variables correspondant à:

- une thématique,
- un pas de temps,
- une structuration des données
- ...

Chaque ligne peut être identifiée par sous-ensemble de colonnes Cette permet de créer ou de mettre à jour une donnée, selon qu'elle est ou non déjà présente en base.

Chaque ligne porte sur une ou plusieurs colonnes une information de temporalité.

Chaque ligne porte aussi, sur une ou plusieurs colonnes, des informations sur le contexte d'acquisition des variables des autres colonnes.

On peut vouloir aussi faire figurer dans la base de données certaines informations non présentes dans le fichier de données.

- des informations liées aux variables que l'on fournit sous la forme de fichier de référentiels (description de site, description de méthodes, description d'unités, description d'outils...)
- des informations constantes ne dépendant pas du fichier (par exemple l'unité de la variable)
- des informations constantes pour l'ensemble du fichier (par exemple le site correspondant aux valeurs du fichier). Ces informations pouvant être décrites dans un cartouche, avant l'en-tête de colonne ou juste sous l'en-tête de colonne (valeur minimum ou maximum)
- des informations calculées à partir d'informations du fichier, d'informations des référentiels déjà déposés ou même des données déjà publiées.

## exemple

supposons que l'on ait un fichier de données météorologiques

```
Région;Val de Loire;;;
Période;06/2004;;;
Date de mesure;Site;Précipitation;Température moyenne;Température minimale;Température maximale
01/06/2004;0s1;30;20;10;24
07/06/2004;0s1;2;22;14;27
07/06/2004;0s2;0;21;9;28
```

- La temporalité est portée par la colonne "Date de mesure"
- Le contexte est portée par l'information du cartouche d'en-tête "Région" et la colonne "Site".
- On identifie 4 variables:
  - date au format dd/MM/yyyy (format au sens SQL : <https://www.postgresql.org/docs/current/functions-formatting.html#FUNCTIONS-FORMATTING-DATETIME-TABLE>). Cette variable n'a qu'une seule composante "day". On note que les moyennes sont calculées à la journée.
  - localization qui fait référence à un site de la colonne "Site", avec deux composantes (site et region)
  - precipitation qui correspond à la pluviométrie de la colonne "Précipitation" avec deux composantes (value,unit=mm)
  - temperature qui se réfère aux colonnes "Température moyenne", "Température minimale" et "Température maximale" avec 4 composantes (value,min,max,unit=°C)

Du coup on peut aussi définir des référentiels pour préciser ses informations

#### **region.csv**

```
code ISO 3166-2;nom
FR-ARA Auvergne-Rhône-Alpes
FR-BFC Bourgogne-Franche-Comté
FR-BRE Bretagne
FR-CVL Centre-Val de Loire
FR-COR Corse
FR-GES Grand Est
FR-HDF Hauts-de-France
FR-IDF Île-de-France
FR-NOR Normandie
FR-NAQ Nouvelle-Aquitaine
FR-OCC Occitanie
FR-PDL Pays de la Loire
FR-PAC Provence-Alpes-Côte d'Azur
```

#### **site.csv**

```
nom:Date de création;region
0s1;01/01/2000;FR-CVL
0s2;01/01/2000;FR-CVL
```

Les sites font référence aux régions.

#### **unite.csv**

```
nom;nom_fr;nom_en;code
temperature;Température;Temperature;°C
precipitation;Précipitation;Precipitation;mm
```

Le fait de dire que l'unité d'une donnée fait référence au référentiel unite signifie:

- que l'unité doit être présente dans ce référentiel,
- que l'on ne pourra pas supprimer une unité du référentiel si on y a fait référence.

On aurait pu rajouter des responsables de site et de région, des descriptions des variables, des intervalles de valeurs...

Ainsi nous avons pu faire une analyse de notre domaine et le format des fichiers qui s'y rapportent. Nous pouvons commencer l'écriture du fichier de configuration.

## Vocabulaire

### Clefs et code

Dans un fichier, on définit une ou plusieurs colonnes qui correspondent à la clef d'identification de la ligne. Cette clef naturelle permet lors d'une insertion / suppression de retrouver cette ligne dans la base de données et, si elle est présente, de la mettre à jour. Dans le cas contraire, une nouvelle ligne est créée.

#### code

Pour enregistrer ces clefs dans la base de données, et pour éviter les erreurs, les clefs sont codées. Le code utilisé n'autorise que les chiffres, les lettres minuscules et majuscules ainsi que le caractère souligné (underscore).

Cependant pour permettre une plus grande souplesse, les accents sont supprimés, les majuscules sont remplacées par les minuscules, les espaces sont remplacés par des \_ et les autres caractères sont remplacés par leur nom ASCII en majuscules.

- L'année de départ -> LAPOSTROPHEannee\_de\_depart
- $\mu\text{mol m}^{-2} \text{ s}^{-1}$  -> MICROSIGNmol\_m2\_s1
- $\text{m}^2/\text{m}^2$  -> mSUPERSCRIPTTWOsolidusmSUPERSCRIPTTWO
- °C -> DEGREESIGNc

Ainsi les valeurs élévation, élévation, elevation ou même EléVaTioN renvoient toutes le même code.

Ces transformations sont faites de manière transparente.



Quand on fait référence à un référentiel, que cela soit pour un type de données ou pour un autre référentiel, on utilise la clef naturelle de ce référentiel. Cependant il sera possible de demander la mise en code de la valeur avant de rechercher son existence dans le référentiel de référence.

#### Clef naturelle.

Elle est construite en concaténant les valeurs des différentes colonnes composant la clef. Le signe de concaténation est le double underscore '\_\_'.

- Forme géométrique de la colonie + prisme -> forme\_geometrique\_de\_la\_colonie\_\_prisme
- Ensoleillement + Ensoleillé -> ensoleillement\_\_ensoleille
- Piégeage en montée + Couleur des individus -> piegeage\_en\_montee\_\_couleur\_des\_individus

#### Clef hiérarchique

Elle est construite en concaténant les clefs naturelles de différents référentiels. Le signe de concaténation de la clef hiérarchique est le point '.' (FULLSTOP)

Ainsi si on a une parcelle "1", dans le site "Site 1" du type de site "Site d'étude" :

référentiel	Nom	Clef naturelle	Clef hiérarchique
Type de site	Site d'étude	site_dAPOSTROPHEetude	site_dAPOSTROPHEetude
Site	Site 1	site_dAPOSTROPHEetude__site_1	site_dAPOSTROPHEetudeFULLSTOPsite_dAPOSTROPHEetude__site_1
Parcelle	1	site_dAPOSTROPHEetude__site_1__1	site_dAPOSTROPHEetudeFULLSTOPsite_dAPOSTROPHEetude__site_1FULLSTOP1

## Référentiels

**references:** Un ensemble d'informations permettant de préciser le contexte de la mesure ou de l'observation. En déportant ces informations dans des fichiers **references**, on évite la répétition d'informations. On utilisera la clef d'une information pour y faire référence.

## Types de données

**data** : Un ensemble de données correspondant à une thématique et un format de fichier commun.

**variable** : correspond à un ensemble de données, qualifiant ou se rapportant à une variable de mesure, d'observation, d'informations, de temporalité ou de contexte.

**composent** : un ensemble de valeur qui servent à décrire une variable (valeur, écart type, nombre de mesure; indice de qualité; méthode d'obtention...)

**localisationScope** : Une ou des informations contextuelles (variable-composent) qui font sens pour limiter les autorisations.

**timeScope** : L'information de temporalité d'une ligne faisant sens pour limiter des autorisations à une période.

**dataGroups** : un découpage, sous forme de partitionnement de variables, en un ensemble de groupes de variables (**dataGroups**), pour limiter les droits à la totalité ou à des sous ensembles de variables.

On pourrait dans notre exemple distinguer 3 **dataGroups**:

- informations(date et localization)
- precipitation(precipitation)
- temperature (temperature) Mais on peut aussi faire le choix d'un seul groupe
- all(date,localization,precipitation,temperature) Ou de 4 groupes en découplant informations en date et localization

# Aide fichier à la rédaction du fichier de configuration

---

## La création :

Vous trouverez ci-dessous un exemple de fichier Yaml fictif qui décrit les parties attendues dans celui-ci pour qu'il soit valide. **Attention le format Yaml est sensible** il faut donc respecter l'indentation.

Il y a 5 parties (**sans indentation**) attendues dans le fichier :

- version,
- application,
- references,
- compositeReferences,
- dataTypes

**l'indentation du fichier yaml est très importante.**

## Description du fichier

Informations sur le fichier lui même

### Version de l'analyseur (parser) du fichier de configuration.

Soit version actuelle du site qui est 1 actuellement. Il faut avoir en tête que lorsque l'application évolue et que la version de l'analyseur s'incrémente, le fichier de configuration peut ne plus être valide.

```
version: 1
```


*version n'est pas indenté.*

**on présente l'application avec son nom et la version du fichier de configuration :**


(on commence par la version 1)

S'il y a déjà une application du même nom mais que l'on a fait des modifications dans le fichier on incrémente la version.

```
application:
  name: application_nom
  internationalizationName:
    fr: Ma première application
    en: My first application
  version: 1
```

 Les sections d'internationalisation ne sont pas obligatoires, mais permettent une internationalisation des interfaces.

*application n'est pas indenté. nom et version sont indentés de 1.*

 Vous trouverez le formalisme d'un fichier yaml sur cette [page](#).

Certains éditeurs de texte permettent d'écrire un yaml avec colorisation et mise en relief des erreurs. par exemple l'éditeur de texte ou kate (linux) ou bien Notepad++ (windows)

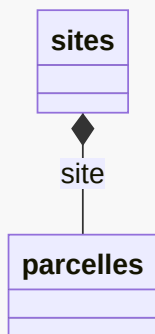
## Description référentiels

on décrit les référentiels dans la partie *references*, on y liste les noms des colonnes souhaitées (dans [columns](#), [computedColumns](#) ou [dynamicColumns](#)); on précisant la liste de colonnes qui forment la clef naturelle (dans [keyColumn](#)). On pourra aussi préciser des règles de validations sur une ou plusieurs colonnes dans la section [validations](#):

- une [columns](#) est une colonne du fichier
- une [computedColumns](#) est une colonne qui n'est pas présente dans le fichier et dont la valeur est une constante ou le résultat d'un calcul.
- une [dynamicColumns](#) est un ensemble de colonnes dont la clef est la concaténation d'un préfixe et d'une valeur d'un référentiel. Par exemple s'il existe un référentiel propriétés avec les valeurs (couleur, catégorie, obligatoire), on pourrait avoir dans un autre référentiel (en utilisant le préfixe "pts\_") pts\_couleur, pts\_catégorie et pts\_obligatoire, en les déclarant comme [dynamicColumns](#).
- 

## Description des colonnes (columns)

Pour le modèle de référentiels



et pour les fichiers :

- **sites.csv**

**nom du site**

---

site1

---

site2

- **parcelles.csv**

**site      nom de la parcelle**

---

site1    1

---

site2    1

on aura le yaml suivant

```

references:
  agroecosystème:
    keyColumns: [nom]
    columns:
      nom:
        nom
  sites:
    #donnée de référence avec une clef sur une colonne
    keyColumns: [nom du site]
    columns:
      Agroecosystème:
        nom du site:
  parcelles:
    #donnée de référence avec une clef sur deux colonnes
    keyColumns: [site,nom de la parcelle]
    columns:
      site:
        nom de la parcelle:

```

⚠ Le nom du référentiel est libre. Cependant, pour ceux réutilisés ailleurs dans l'application, il est préférable de n'utiliser que des minuscules et underscores sous peine de générer des erreurs dans les requête sql ou la création des vues:

exemple: mon\_nom\_de\_referentiel

⚠ Le nom des colonnes des references doivent être courtes pour ne pas être tronqué lors de la création des vues de l'application. Les noms des colonnes dans la base de données est limité à 63 caractères. Dans les vues, ce nom est une concaténation du nom du référentiel et du nom de la colonne

exemple: type\_de\_sites\_\_nom\_du\_type\_de\_site

Pensez à mettre le même nom de colonnes dans le fichier .csv que dans la partie *columns* du fichier yaml.

⚠ *references* n'est pas indenté. *sites* et *parcelles* sont indentés de 1. *keyColumns* et *columns* sont indentés de 2. Le contenu de *columns* seront indenté de 3.

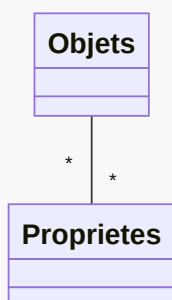
### Colonnes calculées (computed columns)

Une colonne calculée est une colonne qui n'est pas présente dans le fichier. Ses valeurs sont issue du résultat d'un calcul.

```
references:
computedColumns:
  date_iso:
    defaultValue: > #une valeur par défaut qui est une expression groovy (une chaîne
entre cotes "ceci est une valeur par défaut", un nombre, sont des expressions groovy.
    import java.time.LocalDate
    import java.time.format.DateTimeFormatter
    return LocalDate.parse(datum.date, DateTimeFormatter.ofPattern('dd/MM/yyyy'))
      .atStartOfDay()
      .format(DateTimeFormatter.ISO_DATE_TIME)
  checker:
    name: Date
    params:
      pattern: yyyy-MM-ddThh:mm:ss
```

### Colonnes dynamiques (dynamic columns)

Les colonnes dynamiques permettent de traduire une relation n-nentre deux référentiels. par exemple entre un objet et ses propriétés.



Dans le référentiel Propriétés on liste les différentes propriétés qui sont observées sur l'objet

Dans le référentiel Objet, on donne la liste des propriétés observées pour chacune des propriétés dans une colonne avec comme en-tête le nom de la propriété préfixée.



**propriétés.csv**

```

nom de la propriété;isQualitative
couleur:true
nombre_de_faces:false
indice:false

```

**objet.csv**

```

nom de l'objet;pt_couleur;pt_nombre_de_faces;pt_indice
cube;bleu;6;7
tétraèdre;rouge;4;2

```

On définira le référentiel objet de la manière suivante

```

references:
  proprietes;
  columns:
    nom de la propriété
    isQualitative
  keyColumns:[nom de la propriété]
objet;
  columns:
    nom de l'objet
  keyColumns:[nom de l'objet]
  dynamicColumns:
    headerPrefix: pt_ # les colonne commençant par ce préfixe seront comprise comme
étant des colonnes dynamiques
    reference: proprietes #le référentiel qui contient les noms des colonnes
    referenceColumnToLookForHeader: nom de la propriété # la colonne qui contient
les noms des colonnes dans le référentiels sus désigné.
    id: propriétés de l'objet # la clef à utiliser pour enregistrer la "map"
propriété:valeur
    title: propriétés de l'objet # le titre de la colonne en sortie pour lister les
valeurs

```

**On peut poser des contraintes sur les données de référence****Utilisation de vérificateurs (checker)**

Pour chaque colonne on peut ajouter des vérificateurs.

- vérifier la nature d'un champs (float, integer, date) ( Integer, Float, Date)
- vérifier une expression régulière ( RegularExpression)
- ajouter un lien avec un référentiel (Reference)
- verifier un script (le script renvoyant true) ( GroovyExpression)

```

sites:
#donnée de référence avec une clef sur une colonne
  keyColumns: [nom du site]
  columns:
    Agroécosystème:

```

```

nom du site:
  checker:
    name: Reference #contrainte de type référentiel
    params:
      refType: sites #qui porte sur le référentiel site
      required: true # la valeur ne peut être manquante
      transformation:
        codify: true #on transforme la valeur en son code avant de la tester
date:
  checker:
    name: Date
    params:
      pattern: dd/MM/yyyy
      required: true
numéro:
  checker:
    name: Integer

```

#### Utilisation de validations portant sur une ou plusieurs colonnes

Les contraintes se définissent pour chacune des données de référence. Soit dans la définition de la colonne elle même, soit dans la section [validation][#referencesValidation).

Chaque règle de validation peut porter sur plusieurs colonnes de la donnée de référence. Elle comporte une description et un **checker** (Reference, Integer, Float, RegularExpression, Date, GroovyExpression).

```

types_de_donnees_par_themes_de_sites_et_projet:
  validations:
    projetRef: # la clef d'une validation
      internationalizationName:
        fr: "référence au projet" # la description en français
        en: "project reference" # la description en anglais
      checker: # le checker de validation
        name: Reference #Le checker à utiliser
        params: #liste de paramètres (dépend du checker choisi)
          refType: projet #pour le checker référence la donnée référencée
        columns: [nom du projet] #liste des colonnes sur lequel s'applique le checker
    sitesRef:
      internationalizationName:
        fr: "référence au site" # la description en français
        en: "site reference" # la description en anglais
      checker:
        name: Reference
        params:
          refType: sites
        columns: [nom du site]
    themesRef:
      internationalizationName:
        fr: "référence au thème" # la description en français
        en: "thematic reference" # la description en anglais
      checker:
        name: Reference
        params:
          refType: themes
        columns: [nom du thème]

  checkDatatype:

```

```

internationalizationName:
  fr: "existence du type de données" # la description en français
  en: "existence of the data type" # la description en anglais
checker:
  name: GroovyExpression # utilisation d'un script groovy de validation
  params:
    groovy:
      expression: >
        String datatype = Arrays.stream(datum.get("nom du type de
données").split("_")).collect{it.substring(0, 1)}.join();
        return application.getDataType().contains(datatype);
checkDateFormat:
  internationalizationName:
    fr: "date au format dd/MM/yyyy" # la description en français
    en: "date in dd/MM/yyyy format" # la description en anglais
  checker:
    name: Date
    params:
      pattern: dd/MM/YYYY
    columns : [Date de début, Date de fin] # les colonnes du référentiel
concernées par la vérification.

```

## Vérificateurs

Contenu de la section params:

name	References	Integer	Float	Date	GroovyExpression	RegularExpression	*
refType	X						Le référentiels de jointure
pattern						X	Le pattern pour une expression régulière
transformation	X	X	X	X	X	X	La définition d'une transformation à faire avant de vérifier la valeur
required	X	X	X	X	X	X	La valeur ne peut être nulle (true)
multiplicity	X						La colonne contient un tableau de référence (true)
groovy					X		La définition d'une expression groovy

name	References	Integer	Float	Date	GroovyExpression	RegularExpression	*
duration				X			Pour une date la durée de cette date

 Une durée est définie au sens SQL d'un [interval](#) ('1 HOUR', '2 WEEKS', '30 MINUTES').

On peut rajouter une section [transformations](#) pour modifier la valeur avant sa vérification :

Cette transformation peut être configurée avec

- **codify** : la valeur sera alors échappée pour être transformée en clé naturelle (Ciel orangée -> ciel\_orange)
- **groovy** : permet de déclarer une transformation de la valeur avec une expression Groovy (qui doit retourner une chaîne de caractère)

La section groovy accepte trois paramètres

- **expression** : une expression groovy (pour le checker GroovyExpression doit renvoyer true si la valeur est valide)
- **references** : une liste de référentiels pour lesquels on veut disposer des valeurs dans l'expression
- **datatypes** : une liste de datatypes pour lesquels on veut disposer des valeurs dans l'expression

**!alert:** La différence entre une section groovy de la section params d'un checker **groovy** et une section groovy de la section transformation de la section params, tient dans le fait que pour un checker groovy l'expression renvoyée est un booléen tandis que dans la transformation l'expression groovy renvoie une nouvelle valeur.

Pour les checkers GroovyExpression et les transformation Groovy, on récupère dans le script des informations :

```
datum : les valeurs de la ligne courante.
  On récupère la valeur d'un variable-component -> datum.get("nom de la
variable").get("nom du composant")
application : le yaml de l'application
references: les valeurs d'une donnée de référence spécifique;
  Il faut renseigner dans params la clef "references" qui définit les données de
références accessibles dans references.
  -> references.get("nom de la reference").getRefValues().get("nom de la colonne")
referencesValues : idem que references;
  -> referencesValues.get("nom de la reference").get("nom de la colonne")
datatypes : idem que references pour les datatypes. Il faut renseigner le param
datatypes
  -> datatypes.get("nom du datatype").getValues().get("nom de la colonne")
datatypesValues : idem que datatypes
  -> datatypesValues.get("nom du datatype").get("nom de la colonne")
```

 On peut aussi passer des constante dans le script

```
expression : >
import java.time.LocalDate
import java.time.format.DateTimeFormatter

LocalDate minDate = LocalDate.of(2014,1,1)
LocalDate maxDate = LocalDate.of(2022,1,1)
LocalDate date = LocalDate.parse(datum.date,
DateTimeFormatter.ofPattern('dd/MM/yyyy'))
return date.isBefore(maxDate) && date.isAfter(minDate)
```


## <a id="compositeReferences"" />Définition de clefs composites entre différentes références

Une clef composite permet de définir une hiérarchie entre différentes données de référence.

Dans l'exemple ci-dessous il y a une relation oneToMany entre les deux données de référence sites et parcelles.

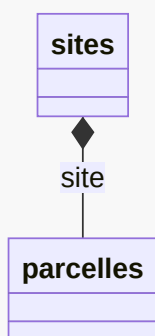
La [clef naturelle](#) permet de distinguer deux lignes distinctes. Elle est juste construite à partir de la concaténation des valeurs de colonnes.

La clef composite rajoute une hiérarchie entre les données de référence. Dans l'exemple ci-dessous pour référencer une ligne site on utilise sa clef naturelle **site1\_\_1**, une clef hiérarchique est aussi créée : **site1.site1\_\_1**

 On peut créer une clef naturelle sur une colonne dont chaque valeur est unique (une colonne clef technique par exemple), que cette colonne soit donnée par le fichier ou bien calculée.


La clef composite est une concaténation de toutes les clefs naturelles qui la compose (séparateur .) cf le chapitre [code](#)

Pour créer une clef à partir d'une chaîne, on peut utiliser un checker et en renseignant la section codify de params.



```

compositeReferences:
  localizations:
    components:
      - reference: sites
      - reference: parcelles
      parentKeyColumn: "site"
  
```

 *compositeReferences* n'est pas indenté. *localizations* est indenté de 1. *components* est indenté de 2. - *reference* et - *parentKeyColumn* sont indentés de 3. Le *reference* qui est sous *parentKeyColumn* est indenté de 4.

Il est possible de définir une référence composite récursive dans le cas de données de références qui font référence à elle même. En ce cas on utilisera la clef [parentRecursiveKey](#) pour faire référence à la colonne parent du même fichier. C'est d'ailleurs le seul moyen de référencer un référentiel sur lui-même.

```

compositeReferences:
  taxon:
    components:
      - parentRecursiveKey: nom du taxon supérieur
      reference: taxon
  
```

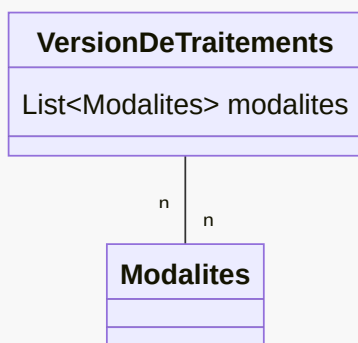
voir aussi la section [autorisations(#authorizations) quand à l'utilisation des clef composites.

### Relation entre deux référentiels avec multiplicité

Lorsqu'un fichier CSV contient une colonne dont le contenu est une liste de clés naturelles pointant vers un autre référentiel, on parle de multiplicité.

On peut configurer un checker de type **Reference** de façon à prendre en compte cette multiplicité.

Par exemple, un fichier CSV de modalités dont la clé naturelle est composée de la seule colonne code :



Une version d'un traitement est définie par une liste de modalités (plus ou moins d'engrais, plus ou moins de pesticide, pature ou non...)

```

Variable de forçage;code;nom_fr;nom_en;description_fr;description_en
Fertilisation;F0;nulle;nulle;Aucune fertilisation;Aucune fertilisation
Utilisation;U0;Sol nu;Sol nu;Maintient du sol en sol nu;Maintient du sol en sol nu
Utilisation;UA;Abandon;Abandon;Pas de traitement;Pas de traitement
Utilisation;UC;Culture;Culture;Utilisation du sol en culture lors d'une rotation;Utilisation du sol en culture lors d'une rotation
Utilisation;UF;Fauche;Fauche;Prairies fauchées;Prairies fauchées
Utilisation;UP;Pâture;Pâture;Prairies pâturées;Prairies pâturées
  
```

accompagné de ce fichier **version\_de\_traitement.csv** :

```

site;traitement;version;date début;date fin;commentaire_fr;commentaire_en;modalites
Theix;T4;1;01/01/2005;;version initiale;initial version;F0,UA
Theix;T5;1;01/01/2005;;version initiale;initial version;F0,UF
  
```

On voit que la colonne **modalites** est multi-valuée : elle contient plusieurs codes vers des clés du fichier modalités.

On paramètre le checker avec la **multiplicity: MANY**. Cela donne, par exemple, un YAML de la forme (voir la section *validations de version\_de\_traitement*) :

```

references:
  modalites:
    keyColumns: [code]
    columns:
      Variable de forçage:
        code:
        nom_fr:
  
```

```

    nom_en:
    description_fr:
    description_en:
  version_de_traitement:
  keyColumns: [site, traitement]
  columns:
    site:
    traitement:
    version:
    date début:
    date fin:
    commentaire_fr:
    commentaire_en:
  modalites:
    internationalizationName:
      fr: "référence aux modalités"
      en: "reference to conditions"
  checker:
    name: Reference
    params:
      refType: modalites
      multiplicity: MANY
      transformation:
        codify: true

```



dans la base, modalites sera un tableau.

## <a id="datatypes"" />Description des *dataTypes*

Pour enregistrer un type de données, il faut déclarer

- le **data** : ce qui sera enregistré en base de données [section data](#)
- le **format du fichier** ([*section format*])(#format)
- les **autorisations** ([section authorizations](#))
- les **validations** de chaque ligne

Nous regrouperons les données par nom des types de données que l'on souhaite importer (nom\_de donnees) correspondant à un format de fichier (*nomDonnée.csv*)



Pour éviter les erreurs, n'utilisez que des minuscules et des \_ dans le nom des types de données. Utilisez la section internationalisationName pour donner un nom plus explicite.

```

dataTypes:
  nom_donnees_csv:
    internationalizationName:
      fr: Le nom des données.
      en: The datatype name.

```

*dataTypes* n'est pas indenté. *nomDonnée* est indenté de 1.

### **data**

La section data permet de décrire le schéma des données enregistrées en base. Les données sont enregistrées comme une liste de *variables* pouvant avoir plusieurs composantes (*components*). Les *variables/components* peuvent être des constantes ou des valeurs calculées, provenir d'un en-tête, ou provenir des colonnes.

*date*, *localization* et *prélèvement* sont des exemples de nom de variable qui regrouperont plusieurs composantes. On fait la liste de *components* pour chaque variable.

Par exemple *day* et *time* sont les composantes (*components*) de la variable *date*.

On vérifie leurs formats grace aux *checker* -> *name* est le nom du checker et *params* permet de définir les paramètres du format via le *pattern*. Voici quelque possibilité de *pattern* possible pour les dates et heures :

pattern	exemple 1	exemple 2
dd/MM/yy	31/01/21	31/12/21
dd/MM/yyyy	31/01/2021	31/12/2021
MM/yyyy	01/2021	12/2021
M/yyyy	1/2021	12/2021
HH:mm	13:00	01:00
hh:mm:ss	13:00:00	01:00:00
dd/MM/yy hh:mm:ss	31/01/21 13:00:00	31/12/21 01:00:00

Pour les dates anglaises inverser le "dd" avec le "MM" (exemple : MM/dd/yy -> 01/31/21) et pour l'heure anglaise il suffit d'ajouter am/pm (exemple "hh:mm am/pm" -> "01:00 am" ou "hh:mm:ss AM/PM" -> "01:00:00 AM"). Le *pattern* doit correspondre avec le format de la date dans le fichier CSV.

pour les données :

date	heure	nom de la parcelle	point	volume	qualité
12/01/2010	10:00:00	site1.site1__1	2	240.7	2
12/01/2010	15:30:00	site2.site2__1	1	105.25	1

On décrit un format pour stocker les données sous la forment

```
{
  date:{
    datetime: "12/01/2010 10:00:00",
    day: "12/01/2010",
    time: "10:00:00"
  },
  localization:{
    parcelle:"site1.site1__1",
    point:"2"
  },
  prélèvement:{
    volume:240.7,
    qualité:2
  }
}
```

```
data:
  date:
    computedComponent: #section pour les composantes calculées
    datetime:
      computation :#calcul d'une valeur par défaut date+time avec une expression
```



```

groovy
    expression: return datum.date.day + " " + datum.date.time
    checker: #ajout d'un checker date dd/MM/yyyy hh:mm:ss
    name: Date
    params:
        pattern: dd/MM/yyyy hh:mm:ss
components: # les composantes non calculées
day:
    checker:
        name: Date
        params:
            pattern: dd/MM/yyyy
time:
    checker:
        name: Date
        params:
            pattern: hh:mm:ss
localization:
    components:
        parcelle:
            checker:
                name: Reference
                params:
                    refType: parcelles
    point:
        checker:
            name: Integer
prélèvement:
    components:
        volume:
            checker:
                name: Float
        qualité:
            checker:
                name: Integer

```



*refType* doit forcément être identique aux noms des références déclarées dans la partie *references*

*data* est indenté de 2. Les variables sont indentés de 3 et les composants le sont de 4.

la validation est utilisé pour valider une ligne sur une ou plusieurs colonnes.

Les *variables/components* sont passés dans la map *datum*. On récupère la valeur du component qualité de la variable SWC

```

validations:
    swcQualityEnumeration:
        localizationName:
            fr: "Si renseignée, la qualité du taux d'humidité vaut 1, 2 ou 3"
            en: "If entered, the quality of the humidity rate is 1, 2 or 3"
        checker:
            name: GroovyExpression
            params:
                groovy:
                    expression: >
                        Set.of("", "0", "1", "2").contains(datum.get("SWC").get("qualité"))

```

Cette formulation vérifie que la valeur du component qualité de la variable SWC est vide ou égale à 0,1 ou 2 L'expression doit renvoyer true

Pour les checkers GroovyExpression, on récupère dans le script des informations :

```
datum : les valeurs de la ligne courante.
  On récupère la valeur d'un variable-component -> datum.get("nom de la
variable").get("nom du composant")
application : le yaml de l'application
references: les valeurs d'une donnée de référence spécifique;
  Il faut renseigner dans params la clef "references" qui définit les données de
références accessibles dans references.
  -> references.get("nom de la reference").getRefValues().get("nom de la
variable").get("nom du composant")
referencesValues : idem que references;
  -> referencesValues.get("nom de la reference").get("nom de la variable").get("nom du
composant")
datatypes : idem que references pour les datatypes. Il faut renseigner le param
datatypes
  -> datatypes.get("nom du datatype").getValues().get("nom de la variable").get("nom
du composant")
datatypesValues : idem que datatypes
  -> datatypesValues.get("nom du datatype").get("nom de la variable").get("nom du
composant")
```

```
unitOfIndividus:
  description: "vérifie l'unité du nombre d'individus"
  checker:
    name: GroovyExpression
    params:
      groovy:
        expression: >
          //definition de constantes
          String codeDatatype= "piegeage_en_montee"
          String codeVariable= "Nombre d'individus"

          /* vérifie que dans le référentiel
variables_et_unites_par_types_de_donnees, la ligne
          ayant comme "nom du type de données" la valeur "piegeage_en_montee" et
comme "nom de la variable"
          la valeur "Nombre d'individus" a dans sa colonne "nom de l'unité" la
valeur du composant "component"
          de la variable "variable" */

          String codeVariable= "Nombre d'individus"
          return
referencesValues.get("variables_et_unites_par_types_de_donnees")
          .findAll{it.get("nom du type de données").equals(codeDatatype)}
          .find{it.get("nom de la variable").equals(codeVariable)}
          .get("nom de l'unité").equals(datum.variable.component);
        references:
          - variables_et_unites_par_types_de_donnees # on joint le contenu du
référentiel variables_et_unites_par_types_de_donnees au contexte.
```

Des valeurs peuvent être définies dans l'expression.

La partie validation peut être utilisée pour vérifier le contenu d'une colonne d'un fichier de données

*validations* est indenté de 2.

### Déclaration des contraintes d'unicité

Il s'agit de déclarer comment une ligne d'un fichier s'exprime de manière unique (contrainte d'unicité au sens de la base de données)

Il ne peut y avoir qu'une seule contrainte d'unicité. Il suffit de déclarer la contrainte dans la section *uniqueness*, en listant la liste des *variable components* qui composent la clef.

Si un fichier possède des lignes en doublon avec lui-même il sera rejeté.

Si une ligne possède la même clef qu'une ligne de la base de données, la ligne sera mise à jour.

Les contraintes ne s'appliquent que pour les fichiers d'un même type de données.

Exemple de déclaration de deux contraintes portant respectivement sur 3 et 2 valeurs.

```
dataTypes:
  mon_datatype:
    uniqueness:
      - variable: projet
        component: value
      - variable: site
        component: chemin
      - variable: date
        component: value
```

**ensuite on va décrire le format des données attendues (dans *format*) décrite dans la partie *dataTypes* :**

Cette section permet de faire le lien avec des informations du fichier et les différentes composantes de variables définies dans la section *data*. On peut y lier aux composantes des *constantes*, des *colonnes* ou même un modèle de *colonnes répétées*.

On précisera aussi l'emplacement de l'en-tête (**headerLine**), de la première ligne de données (**firstRowLine**), et éventuellement du séparateur de champs (**separator** valeur par défaut "")

### Définition de constantes

Si votre fichier à des données mise dans un cartouche vous devrez les décrire dans la partie *constants*. On précisera le nombre de lignes dans la cartouche dans *rowNumber* et le nombre de colonnes utiliser dans la cartouche dans *columnNumber*. On peut aussi choisir pour des informations sous l'en-tête de préciser le nom de l'en-tête *headerName* en lieu et place du numéro de colonne.

ici le contenu de la première ligne deuxième colonne est lié au variable/component localization/nomDonnée et apparaîtra à l'export comme une colonne "type de données"

```
format:
  constants:
    - rowNumber: 1
      columnNumber: 2
      boundTo:
        variable: localization
```

```
    component: nomDonnée
    exportHeader: "type de données"
```

*format est indenté de 2.*

*headerLine* permet de mettre le nombre de la lignes qui contient le nom des colonnes décrite plus bas dans *columns*.

```
headerLine: 1
```

*firstRowLine* sera égale au numéro de la première ligne dans la quelle se trouvera les premières données.

```
firstRowLine: 2
```

Si l'on veut faire référence à des lignes entre la ligne d'en-tête et la première ligne de données, on peut faire référence à la colonne par le nom de l'en-tête de colonne plutôt que par le numéro de la colonne. En ce cas on utilise le champs *headerName*.

```
- rowNum: 11
  headerName: H2O
  boundTo:
    variable: H2O
    component: max_value
  exportHeader: "H2O_max"
```

*headerName* doit avoir exactement le même nom que le nom de la colonne dans le fichier csv.

#### Lien avec les colonnes

*columns* est la partie dans laquelle nous décrivons comment les colonnes sont liées aux composantes de variables (pour l'exemple utilisé ici c'est pour les données du fichier nomDonnées.csv):

```
columns:
- header: "nom de la parcelle"
  boundTo:
    variable: localization
    component: parcelle
- header: "point"
  boundTo:
    variable: localization
    component: point
- header: "date"
  boundTo:
    variable: date
    component: day
- header: "heure"
  boundTo:
    variable: date
    component: time
- header: "volume"
  boundTo:
```

```

    variable: prélèvement
    component: volume
- header: "qualité"
  boundTo:
    variable: prélèvement
    component: qualité

```

#### Lien avec les colonnes répétées

Il est possible d'utiliser un template lorsque certaines colonnes de datatype on un format commun. par exemple avec des colonnes dont le nom répond au pattern `variable_profondeur_répétition : SWC_{[0-9]}_{[0-9]}`

Date	Time	SWC_1_10	SWC_2_10	SWC_3_10	SWC_4_10
01/01/2001	01:00	45	35	37	49
01/01/2001	02:00	45	35	37	49

Il est possible d'enregistrer toutes les colonnes `SWC_{[0-9]}_{[0-9]}` dans une variable unique `swc`.

On declare cette variable dans la section data

```

SWC:
  components:
    variable:
      checker:
        name: Reference
        params:
          refType: variables
          required: true
          codify: true
    value:
      checker:
        name: Float
        params:
          required: false
    unit:
      defaultValue:
        expression: return "percentage"
      checker:
        name: Reference
        params:
          refType: unites
          required: true
          codify: true
    profondeur:
      checker:
        name: Float
        params:
          required: true
    repetition:
      checker:
        name: Integer
        params:
          required: true

```

Dans la section `format` on rajoute une section *repeatedColumns* pour indiquer comment remplir le data à partir du pattern

```
format:
  ...
  repeatedColumns:
    - headerPattern: "(SWC)_([0-9]+)_([0-9]+)"
      tokens:
        - boundTo:
            variable: SWC
            component: variable
            exportHeader: "variable"
        - boundTo:
            variable: SWC
            component: repetition
            exportHeader: "Répétition"
        - boundTo:
            variable: SWC
            component: profondeur
            exportHeader: "Profondeur"
      boundTo:
        variable: SWC
        component: valeur
        exportHeader: "SWC"
```

On note la présence de la section `token` contenant un tableau de `boundTo` dans lequel le résultat des capture de l'expression régulière seront utilisés comme une colonne. token d'indice 0 -> \$1 token d'indice 1 -> \$2

etc...

Dans l'exemple le variable-component SWC-variable aura pour valeur SWC résultat de la première parenthèse.

**authorization** Dans la section *authorization*, on définit les objets sur lesquels porteront les les autorisations d'accès aux données :

Authorization permet de définir des groupes de variables. Une ligne du fichier est découpée en autant de ligne que de *dataGroups*. On définit aussi des composantes de portée: *authorizationScope* et la composante temporelle : *timeScope*. Les droits sont portés par la ligne. (un *dataGroup* + un *authorizationScope* + un *timeScope*)

#### Groupe de variables (datagroups)

Une fois définie toutes les variables, on imagine un découpage de celles-ci faisant sens. Pour chaque groupe ainsi défini, on pourra ou non accorder les droits, et ce, indépendamment des autres groupes. Un groupe comprends des variables corrélées (une valeur + une moyenne + un nombre d'observation + un écart-type + une unité + une méthode...). On pourra aussi regrouper des variable de contexte (site, plateforme) ou temporelles (date, durée)

#### Portée des données (authorizationScope).

Il s'agit là de définir un ensemble de composantes que l'on pourra sélectionner dans un arbre, pour limiter la portée de l'autorisation. Pour que l'interface puisse proposer des choix de portée, il est nécessaire que toutes les composantes citées dans *authorizationScope* soient liées à un référentiel avec une section *checker* de type *References*. Pour limiter le nombre d'entrées dans l'arbre de portée, il convient de définir dans la section *compositeReferences* comment les différentes composantessont liées entre elles. Le cas échéants, une combinaison des différentes composantes sera faite.

**Temporalité des données (timeScope).**

On définit une composante portant une information de temporalité. Elle définira la portée temporelle de la ligne. Cette composante doit nécessairement être liée à un checker de type Date.

Certains patterns de date définissent une durée par défaut.

pattern	durée de la période par défaut
yyyy	1 an
MM/yyyy	1 mois
dd/MM/yyyy	1 journée
dd/MM/yyyy HH:mm:ss	1 journée
tous les autres	1 journée

Il est possible de forcer la durée d'un date en précisant la **duration** dans le checker (1 DAY, 30 MINUTES)

Vous pouvez préciser la durée du timescope dans le params "duration" au format:

- ([0-9]\*) (NANOS|MICROS|MILLIS|SECONDS|MINUTES|HOURS|HALF\_DAYS|DAYS|WEEKS|MONTHS|YEARS)

```
authorization:
  dataGroups:
    typeDonnée1:
      label: "Référentiel"
      data:
        - date
        - localization
    typeDonnée2:
      label: "Données qualitatives"
      data:
        - prélèvement
  authorizationScopes:
    localization_ref1:
      variable: localization
      component: parcelle
    localization_ref2:
      variable: localization
      component: point
  timeScope:
    variable: date
    component: datetime
```

```
authorization:
  ...
  timeScope:
    variable: date
    component: datetime

data:
  date:
    components:
      datetime:
        checker:
```

```

name: Date
params:
  pattern: dd/MM/yyyy HH:mm:ss
  duration: 30 MINUTES

```

*authorization* est indenté de 2. *dataGroups*, *authorizationScopes* et *timeScope* sont indenté de 3.

## lors de l'importation du fichier yaml :

- mettre le nom de l'application en minuscule,
- sans espace,
- sans accent,
- sans chiffre et
- sans caractères spéciaux

## Internationalisation du fichier yaml:

Il est possible de faire un fichier international en ajoutant plusieurs parties Internationalisation en précisant la langue.

### Internationalisation de l'application:

Dans la partie application ajouter *defaultLanguage* pour préciser la langue par default de l'application. Ainsi que *internationalization* qui contient les abbreviations des langues de traduction (ex: *fr* ou *en*) Ce qui permettra de traduire le nom de l'application.

```

defaultLanguage: fr
internationalization:
  fr: Application_nom_fr
  en: Application_nom_en

```

### Internationalisation des *references*:

Nous pouvons faire en sorte que le nom de la référence s'affiche dans la langue de l'application en y ajoutant *internationalizationName* ainsi que les langues dans lequel on veut traduire le nom de la référence. *internationalizedColumns* .....

```

references:
  especes:
    internationalizationName:
      fr: Espèces
      en: Species
    internationalizedColumns:
      esp_definition_fr:
        fr: esp_definition_fr
        en: esp_definition_en

```

- Définition d'un affichage d'un référentiel'

Il est possible de créer un affichage internationalisé d'un référentiel (dans les menus, les types de données). Pour cela on va rajouter une section *internationalizationDisplay*.

```

internationalizationDisplay:
  pattern:

```



```
fr: '{nom_key} ({code_key})'
en: '{nom_key} ({code_key})'
```

On définit un pattern pour chaque langue en mettant entre accolades les nom des colonnes. C'est nom de colonnes seront remplacés par la valeur de la colonne ou bien, si la colonne est internationalisée, par la valeur de la colonne internationalisée correspondant à cette colonne.

Par défaut, c'est le code du référentiel qui est affiché.

### Internationalisation des *dataTypes*:

Nous pouvons aussi faire en sorte que *nomDonnéeCSV* soit traduit. Même chose pour les noms des *dataGroup*.

```
dataTypes:
  nomDonnéeCSV:
    internationalizationName:
      fr: Nom Donnée CSV
      en: Name Data CSV
    authorization:
      dataGroups:
        référentiel:
          internationalizationName:
            fr: Référentiel
            en: Referential
          label: "Référentiel"
          data:
            - date
            - projet
            - site
            - commentaire
```

On peut surcharger l'affichage d'une colonne faisant référence à un référentiel en rajoutant une section *internationalizationDisplay* dans le *dataType*.

```
pem:
  internationalizationDisplay:
    especes:
      pattern:
        fr: 'espèce :{esp_nom}'
        en: 'espèce :{esp_nom}'
```

## Zip de YAML

Il est possible au lieu de fournir un yaml, de fournir un fichier zip. Cela permet de découper les YAML long en plusieurs fichiers.

Dans le zip le contenu de la section

<sous\_section><sous\_sous\_section> sera placé dans un fichier sous\_sous\_section.yaml que l'on placera dans le dossier sous\_section du dossier section.

Au premier niveau il est possible de placer un fichier configuration.yaml qui servira de base à la génération du yaml. A défaut de se fichier on utilisera comme base

```
version: 1
```

voici un exemple du contenu du zip :

```
multiyaml.zip
├─ application.yaml
├─ compositeReferences.yaml
├─ configuration.yaml
├─ dataTypes
│   ├── smp_infracj.yaml
│   └─ ts_infracj.yaml
└─ references
    └─ types_de_zones_etudes.yaml
```

## lors de l'importation du fichier yaml :

- mettre le nom de l'application en minuscule,
- sans espace,
- sans accent,
- sans chiffre et
- sans caractères spéciaux

## Aide fichier .csv

---

### lors de l'ouverture du fichier csv via libre office:

\* sélectionner le séparateur en ";"

### lors de la création du fichier csv de Référence et de donnée :

- cocher lors de l'enregistrement du fichier
  - Éditer les paramètres du filtre
  - Sélectionner le point virgule
- dans les données qui se trouvent dans les colonnes contenant des clés naturelles on attend :
  - pas d'accents
  - pas de majuscules
  - pas de caractères spéciaux ( ) , - :
  - autorisé les \_ et les .
- le nom des colonnes doit être le plus court possible
- le fichier doit être en UTF8 pour que les colonnes soient lisibles (les caractères spéciaux ne passent pas sinon. ex : é, è, ç)

### lors de l'importation du fichier csv dans l'application:

- ouvrir la console avec F12 dans votre navigateur pour voir l'erreur de téléversement (erreur serveur) plus en détail.