



# Introduction à dbt (data build tool)

« dbt transforme le SQL en code réutilisable, testé et documenté — c'est l'outil qui permet aux équipes data de travailler comme de véritables équipes de développement logiciel. » 

# Origines, philosophie, concepts clés et bénéfices

O1

## Origines & philosophie

Découverte des fondations historiques de dbt

O2

## Stratégie pour Data Engineers

Comprendre l'importance stratégique de dbt

O3

## Bénéfices concrets

Code, collaboration, documentation et lineage

O4

## Concepts clés

Models, matérialisations, macros et tests

O5

## Best practices & CI/CD

Intégration dans les workflows modernes

O6

## Ressources & démo

Mise en pratique et prochaines étapes



# Origines de dbt

1

## 2016 - RJMetrics

Création initiale chez RJMetrics pour résoudre le chaos des transformations SQL ad-hoc et non versionnées

2

## Fishtown Analytics

Développement et structuration de l'outil pour permettre aux analystes d'écrire des transformations SQL propres et testées

3

## dbt Labs

Évolution vers une plateforme mature adoptée massivement par les équipes data modernes



- L'origine pratique de dbt répond à un besoin réel : transformer le SQL chaotique en code maintenable et fiable.



# Philosophie de dbt

## « SQL as code »

Transformer le SQL en code modulable, testable et maintenable, appliquant les principes du développement logiciel aux transformations de données

## Bonnes pratiques du software engineering

Intégration native du contrôle de version, des tests automatisés, de la CI/CD et de la documentation dans les workflows data

## ELT in-warehouse

Transformation directe dans l'entrepôt de données, exploitant la puissance de calcul moderne des plateformes cloud

## Convention over configuration

Structure de projet standardisée et approche déclarative qui réduit la complexité et améliore la cohérence

# Pourquoi dbt est stratégique pour les Data Engineers



## Réduction de la dette technique

Modularité et réutilisabilité qui éliminent la duplication de code et facilitent la maintenance à long terme



## Tests automatisés

Garantie de qualité des datasets par validation automatique des contraintes métier et techniques



## Documentation & lineage automatiques

Visibilité complète des dépendances et impact analysis pour des changements sécurisés



## Collaboration renforcée

Workflow Git avec pull requests et code review qui apporte traçabilité et qualité collaborative



## CI/CD et observabilité

Intégration native dans les pipelines modernes avec monitoring et alerting avancés

# Bénéfices concrets



## Code

Modularité, macros réutilisables et principe DRY (Don't Repeat Yourself) qui transforment le SQL spaghetti en architecture propre et maintenable



## Collaboration

Workflow Git intégré avec pull requests et code review qui permet une collaboration transparente et tracée entre les équipes



## Documentation

Génération automatique de documentation avec lineage graphique interactif et site statique pour l'exploration des données



## Quality & Observabilité

Tests de données, snapshots pour l'audit et logs détaillés qui garantissent la fiabilité et la traçabilité complète



# Documentation & lineage en action

## Graphe de lineage interactif

Visualisation en temps réel des dépendances entre modèles, permettant une compréhension immédiate de l'impact des changements

## Documentation enrichie

Métadonnées automatiques, descriptions des colonnes et recherche avancée pour une exploration intuitive des assets data

- Le lineage graphique devient l'outil indispensable pour remonter la chaîne d'impact lors de modifications critiques ou d'incidents de données.



# Models : le concept fondamental

## Définition d'un model

Un fichier .sql contenant une instruction SELECT que dbt compile, exécute et persiste automatiquement dans l'entrepôt de données

## Organisation structurée

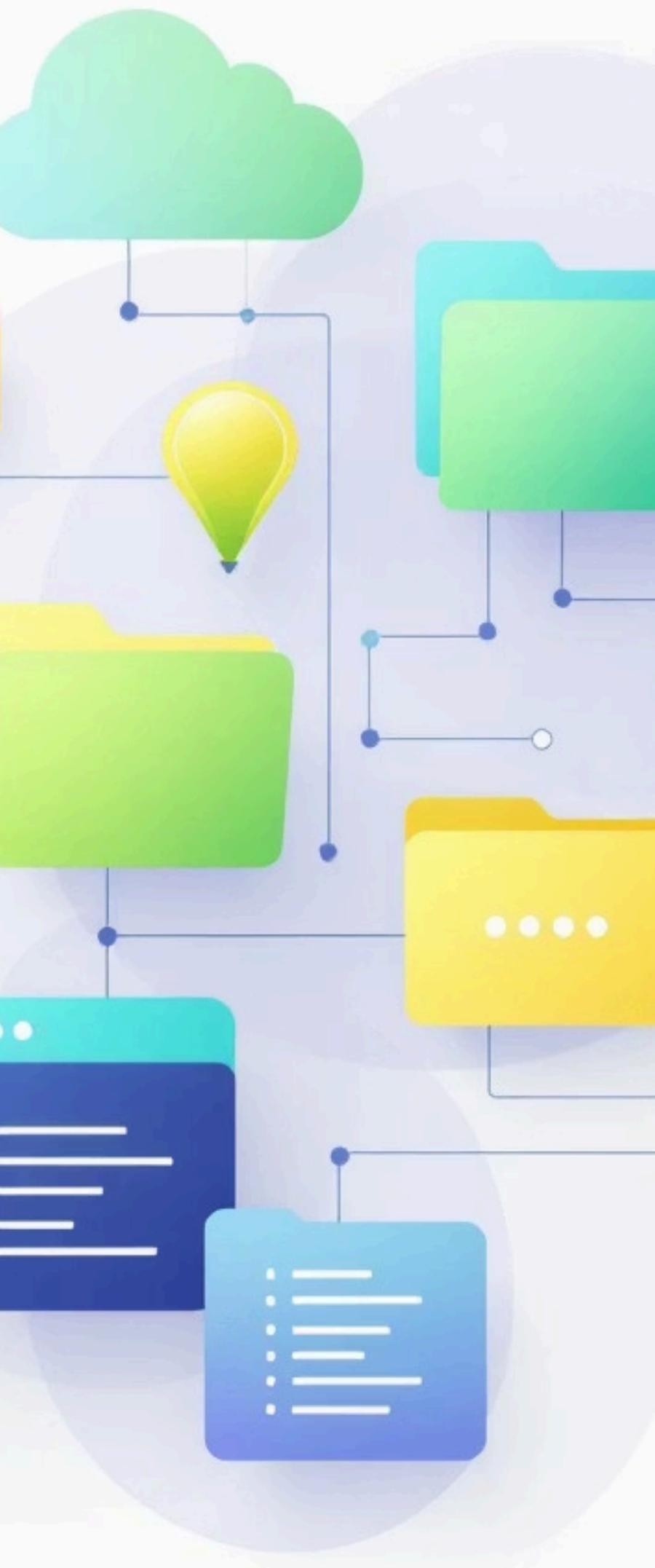
Architecture dans le dossier models/ avec séparation claire : staging pour les données brutes, marts pour les modèles métier finaux

## Exemple concret

Transformation progressive des données raw en insights métier via une chaîne de models interconnectés et documentés

# Structure d'un projet dbt

Un projet dbt est organisé de manière standardisée, facilitant la collaboration et la maintenabilité. Voici les principaux répertoires et fichiers qui constituent l'ossature d'un projet dbt.



## dbt\_project.yml



Le fichier de configuration central du projet, définissant les paramètres globaux, les chemins et les profiles de connexion à la base de données.

## models/



Contient tous les fichiers SQL qui définissent vos transformations de données. Organisé souvent en sous-dossiers (staging, marts) pour une meilleure clarté.

## macros/



Regroupe les blocs de code Jinja réutilisables, équivalents aux fonctions dans les langages de programmation, pour éviter la répétition.

## tests/



Définit les tests personnalisés pour valider la qualité et l'intégrité de vos données, en complément des tests génériques de dbt.

## seeds/



Stocke les fichiers CSV contenant des données statiques ou de référence qui peuvent être chargées directement dans l'entrepôt de données.

## analyses/



Héberge les requêtes SQL ad-hoc qui ne sont pas matérialisées, utiles pour l'exploration de données ou la création de rapports ponctuels.

# Commandes de base

## dbt

Découvrez les commandes fondamentales pour construire, tester et documenter votre projet dbt, éléments clés de votre pipeline de données.



### dbt run

Exécute vos modèles SQL, transformant les données sources en tables ou vues dans votre entrepôt.



### dbt test

Valide la qualité et l'intégrité de vos données en exécutant les tests définis sur vos modèles.



### dbt docs generate & serve

Génère la documentation de votre projet avec un graphe de lineage interactif et un site web explorateur.



### dbt seed

Charge des fichiers CSV contenant des données statiques ou de référence directement dans votre entrepôt.



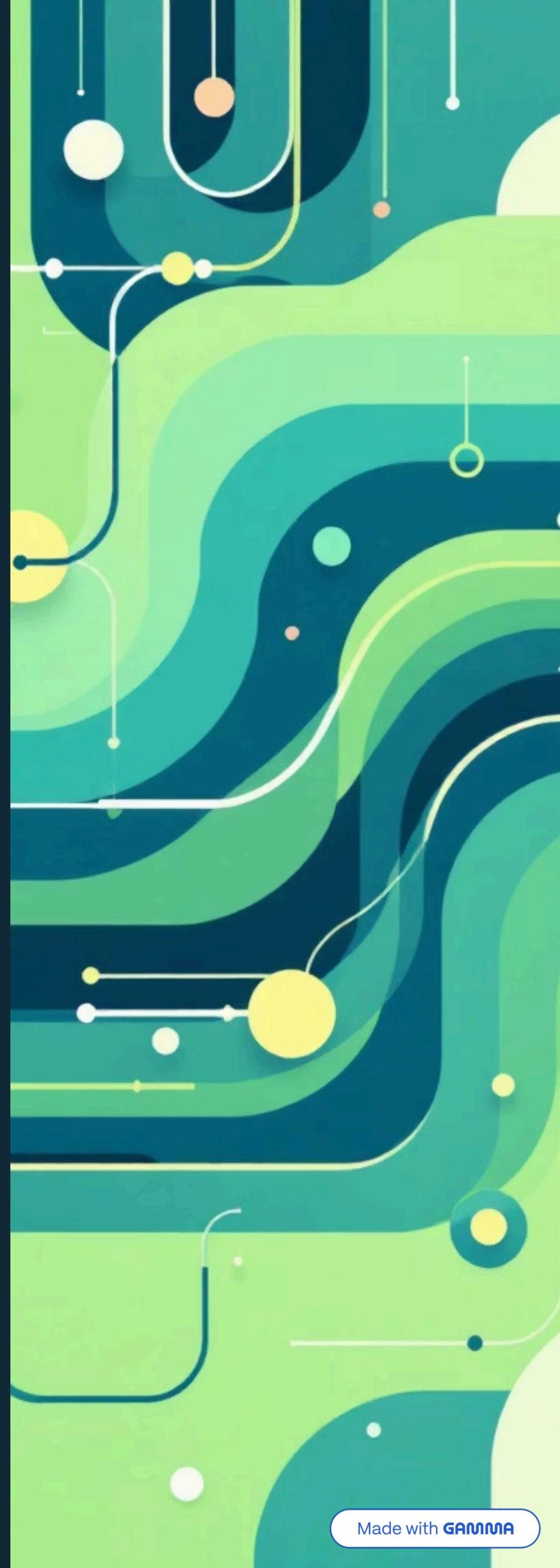
### dbt build

Combine les commandes `run`, `test` et `snapshot` pour une exécution complète et fiable du pipeline.



### dbt debug

Fournit des informations de diagnostic sur la configuration de votre projet et la connectivité à la base de données.



# Comment écrire un modèle dbt

Un modèle dbt est avant tout un fichier SQL qui contient une instruction SELECT. Il décrit une transformation de données et comment les données brutes ou semi-transformées doivent être agrégées, filtrées ou jointes pour créer un nouvel ensemble de données.

```
-- models/marts/core/fact_sales.sql
{{ config(
    materialized='table',
    tags=['fact', 'finance']
) }}

WITH sales_data AS (
    SELECT
        order_id,
        customer_id,
        product_id,
        order_date,
        quantity,
        price_usd
    FROM
        {{ source('raw_data', 'sales') }}
    WHERE
        order_date >= '2023-01-01'
),

customers AS (
    SELECT
        customer_id,
        customer_segment
    FROM
        {{ ref('stg_customers') }}
)

SELECT
    sd.order_id,
    sd.customer_id,
    c.customer_segment,
    sd.product_id,
    sd.order_date,
    sd.quantity,
    sd.price_usd,
    sd.quantity * sd.price_usd AS total_revenue
FROM
    sales_data sd
JOIN
    customers c ON sd.customer_id = c.customer_id
WHERE
    sd.total_revenue > 0
```

## → SQL Standard

Chaque modèle est basé sur une requête SELECT standard, définissant la logique de transformation.

## → Jinja templating

Utilisation de Jinja ({{ ... }}) pour rendre les modèles dynamiques et réutilisables, intégrant logiques et macros.

## → Fonctions ref() et source()

Ces fonctions permettent de référencer d'autres modèles dbt (ref) ou des tables sources (source), créant ainsi un graphe de dépendances clair.

## → Configurations (config())

Définit des propriétés comme le type de matérialisation (vue, table, incrémental) et les tags, pour un contrôle granulaire.

# Types de Matérialisation

dbt offre plusieurs stratégies de matérialisation pour vos modèles, déterminant comment les requêtes SQL sont exécutées et persistées dans votre entrepôt de données. Le choix dépend de vos besoins en performance, coûts et fraîcheur des données.

## View (Vue)

Une vue est une requête SQL sauvegardée qui s'exécute à chaque interrogation. Idéale pour les modèles simples ou les données qui doivent être toujours à jour, sans coût de stockage supplémentaire.

## Table (Table)

Une table matérialise le résultat de votre requête SQL dans une table physique. Excellente pour les modèles complexes, les données volumineuses ou lorsque la performance est critique, mais avec des coûts de stockage et de rafraîchissement.

## Incremental (Incrémentale)

Cette stratégie matérialise votre modèle en insérant ou mettant à jour uniquement les nouvelles données depuis la dernière exécution. Parfait pour les grands ensembles de données qui changent fréquemment, réduisant le temps de traitement et les coûts.

## Ephemeral (Éphémère)

Les modèles éphémères ne sont pas matérialisés dans la base de données. Ils sont compilés en tant que CTEs (Common Table Expressions) et intégrés directement dans les requêtes des modèles en aval, optimisant la clarté du code sans impact sur la performance de la base.



# Sources de données et tests

dbt ne se contente pas de transformer vos données, il vous aide aussi à gérer et valider leur intégrité dès la source.



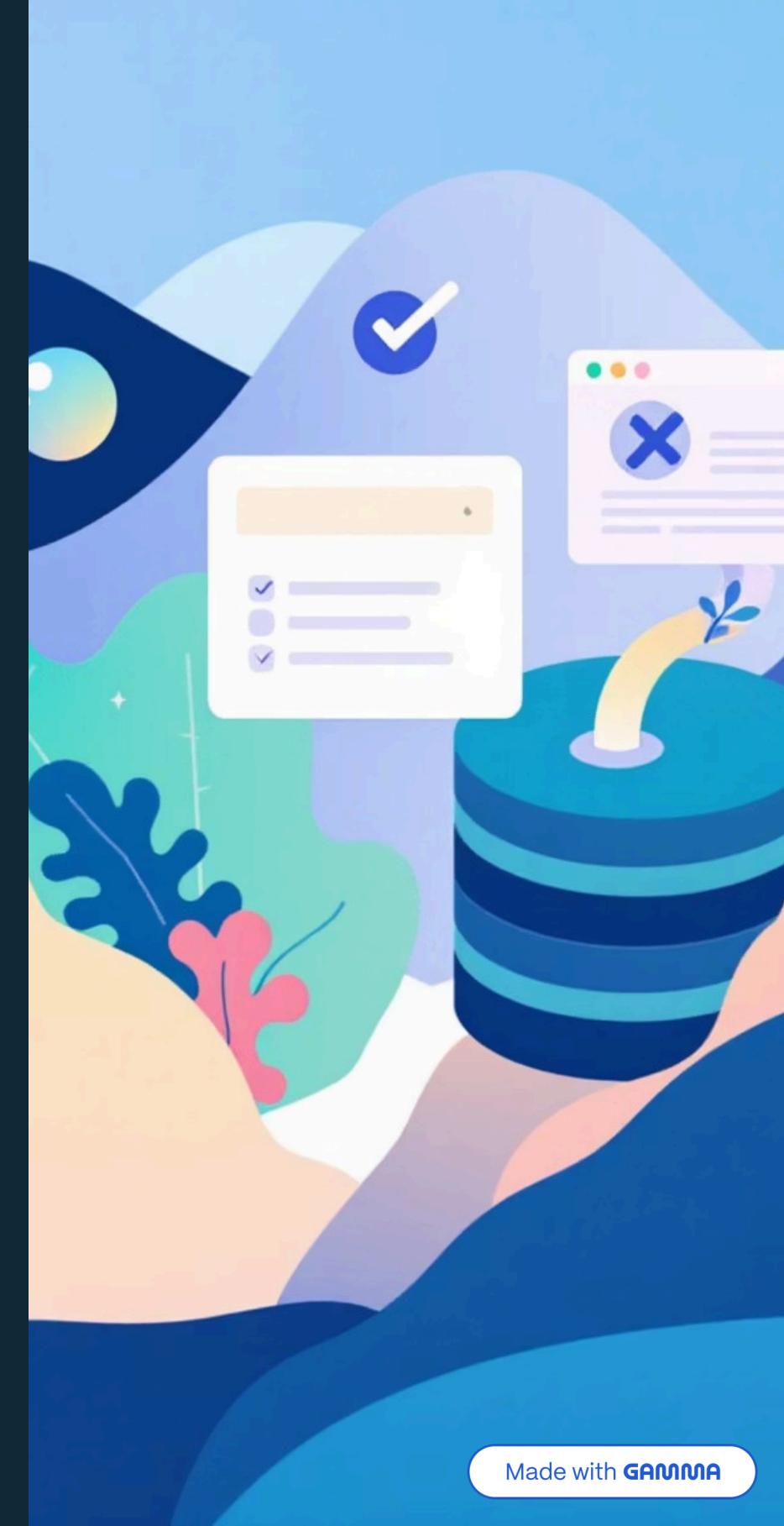
## Déclaration des sources

Définissez vos tables de données brutes via des fichiers `sources.yml`. Cela permet à dbt de suivre le lineage et de générer la documentation automatiquement, en utilisant `{{ source('schema', 'table') }}` dans vos modèles.



## Garantie de qualité avec les tests

Assurez la fiabilité de vos données avec des tests intégrés (`not_null`, `unique`, `accepted_values`, `relationships`) ou personnalisés, définis dans les fichiers `schema.yml`. Détectez les anomalies avant qu'elles n'impactent vos rapports.



# Mise en Pratique : Sources et Tests dbt

Voyons comment concrètement déclarer vos sources de données et définir des tests robustes pour garantir la qualité de vos pipelines.

## Déclaration des Sources (schema.yml)

Définissez vos sources de données externes dans un fichier `schema.yml` pour permettre à dbt de comprendre leur structure et leurs dépendances. Cela est essentiel pour la traçabilité (lineage) et la documentation automatique.

```
version: 2

sources:
  - name: raw_data # Nom de votre source
    database: my_database # Base de données (optionnel, hérite du profil)
    schema: public # Schéma où résident vos tables brutes
    tables:
      - name: sales # Table source
        description: Données brutes des ventes
        columns:
          - name: order_id
            description: Identifiant unique de la commande
        tests:
          - unique
          - not_null
      - name: order_date
        description: Date de la commande
        tests:
          - not_null
      - name: customers
        description: Données brutes des clients
```

## Tests de Qualité sur les Sources et Modèles

Les tests sont définis dans le même fichier `schema.yml` et s'appliquent aussi bien aux sources qu'aux modèles dbt. Ils sont cruciaux pour détecter les anomalies tôt dans le processus.

```
version: 2

models:
  - name: stg_customers # Exemple de modèle
    description: Clients mis en scène
    columns:
      - name: customer_id
        description: Clé primaire du client
    tests:
      - unique
      - not_null
  - name: customer_segment
    description: Segment de clientèle
    tests:
      - accepted_values:
          values: ['VIP', 'Standard', 'New']

  - name: fact_sales
    description: Table de faits des ventes
    columns:
      - name: customer_id
    tests:
      - relationships:
          to: ref('stg_customers') # Test de relation avec un autre modèle
          field: customer_id
```

# Macros dbt : l'automatisation à votre service

Les macros dbt sont des gabarits Jinja qui vous permettent d'écrire du SQL réutilisable, agissant comme des fonctions dans d'autres langages de programmation. Elles sont essentielles pour rendre votre code DRY (Don't Repeat Yourself), garantir la cohérence et simplifier des logiques complexes.

## Réutilisabilité

Écrivez une fois, utilisez partout. Évitez la duplication de code dans plusieurs modèles.

## Cohérence

Appliquez une logique standard (ex: formatage de dates, colonnes d'audit) de manière uniforme.

## Abstraction

Encapsulez des logiques SQL complexes dans des appels de macro simples.

## Génération Dynamique

Générez du SQL basé sur des paramètres d'entrée ou des conditions spécifiques.

## Exemple : Macro pour colonnes d'audit

Un cas d'usage courant est l'ajout de colonnes d'audit comme `created_at` et `updated_at`. Créons une macro simple pour cela.

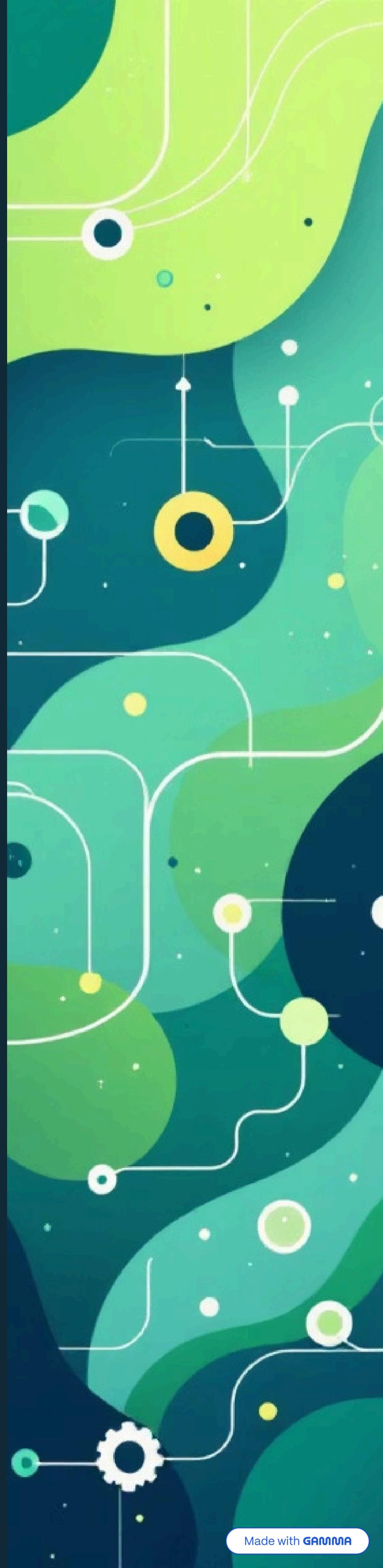
```
-- macros/get_audit_columns.sql

{% macro get_audit_columns() %}
    created_at as created_timestamp,
    updated_at as updated_timestamp
{% endmacro %}
```

Pour l'utiliser dans un modèle, il suffit d'appeler la macro à l'intérieur de votre instruction SELECT.

```
-- models/my_model.sql

SELECT
    id,
    name,
    {{ get_audit_columns() }} -- Appel de la macro
FROM
    raw_data.my_table
```



# Les Packages dbt : Étendre les Capacités de votre Projet

Les packages dbt sont des dépôts de projets dbt pré-construits et partageables qui contiennent des modèles, des macros, des tests et d'autres configurations dbt. Ils permettent de réutiliser du code développé par la communauté ou par votre propre équipe pour résoudre des problèmes courants ou implémenter des logiques spécifiques.

## Pourquoi utiliser des packages dbt ?

- **Réutilisabilité** : Ne réinventez pas la roue. Les packages offrent des solutions prêtes à l'emploi pour des connecteurs (ex: Fivetran, Segment) ou des modèles analytiques standards (ex: suivi des sessions, attribution marketing).
- **Meilleures pratiques** : Profitez de l'expertise de la communauté dbt, avec du code souvent optimisé et testé, intégrant les meilleures pratiques de modélisation de données.
- **Accélération du développement** : Intégrez rapidement des fonctionnalités complexes sans avoir à écrire tout le code SQL et Jinja depuis zéro.
- **Standardisation** : Assurez une approche cohérente pour des transformations de données similaires au sein de votre organisation ou entre différents projets.

## Installation d'un package

Pour installer un package, créez un fichier nommé `packages.yml` à la racine de votre projet dbt et spécifiez les packages que vous souhaitez utiliser. Vous pouvez trouver des packages sur [dbt Hub](#) ou les référencer directement via un dépôt Git.

```
# packages.yml
packages:
  - package: dbt-labs/dbt_utils
    version: 1.1.1
  - package: fivetran/stripe
    version: 0.14.0
  - git: "https://github.com/mon-organisation/mon-package.git"
    revision: main # ou une version spécifique, un tag, un hash de commit
```

Après avoir configuré `packages.yml`, exécutez la commande suivante dans votre terminal pour télécharger les packages :

```
dbt deps
```

## Utilisation des packages dans vos modèles

Une fois installés, les packages sont accessibles comme des parties intégrantes de votre projet. Vous pouvez référencer leurs modèles et leurs macros dans votre propre code.

```
-- Exemple d'utilisation d'un modèle d'un package
SELECT * FROM {{ ref('stripe', 'stripe_charges') }}

-- Exemple d'utilisation d'une macro d'un package
SELECT
  order_id,
  {{ dbt_utils.surrogate_key(['customer_id', 'order_date']) }} as order_key,
  amount
FROM
  {{ ref('my_sales_model') }}
```



# Documentation et Métadonnées

dbt vous permet de créer une documentation riche et directement intégrée à votre projet, rendant vos modèles de données compréhensibles et maintenables par tous les membres de l'équipe.

## Fichiers ``schema.yml``

C'est l'endroit principal pour décrire vos modèles, leurs colonnes, et définir les tests associés. Ces informations sont la base de votre documentation de données.

## Fichier ``dbt_project.yml``

Utilisez ce fichier pour ajouter des descriptions générales à votre projet et à ses composants. C'est le point de départ pour comprendre l'objectif global du projet.

## Fichiers ``.md``

Intégrez des fichiers Markdown directement dans votre projet, notamment un ``README.md`` pour des explications plus détaillées ou des guides d'utilisation, enrichissant la documentation.

# Bonnes Pratiques dbt

Pour garantir la performance technique, la maintenabilité et l'optimisation des coûts de votre projet dbt, suivez ces principes fondamentaux :



## Performance & Coûts

Choisissez une matérialisation adaptée (incrémental pour grands volumes) et optimisez votre SQL pour des exécutions rapides et économiques.



## Qualité des Données

Mettez en place des tests exhaustifs (unicité, nullité, relations) pour valider l'intégrité des données à chaque étape.



## Documentation Claire

Décrivez vos modèles et colonnes dans `schema.yml` et maintenez un `README.md` détaillé pour faciliter la compréhension.



## Modularité & Réutilisation

Structurez votre projet en petits modèles logiques et utilisez `ref()` pour créer un graphe de dépendances clair et efficace.



# Prêts à transformer votre data stack ?

# L'avenir de l'ingénierie des données

dbt représente bien plus qu'un outil : c'est un changement de paradigme qui élève les équipes data au niveau des équipes de développement logiciel. La combinaison de bonnes pratiques, d'automatisation et de collaboration ouvre la voie à une nouvelle ère de fiabilité et d'efficacité dans l'analytics engineering.

## Documentation officielle

[docs.getdbt.com](https://docs.getdbt.com) pour approfondir

## Communauté active

dbt Slack et ressources partagées

## Formation pratique

dbt Learn pour la montée en compétences