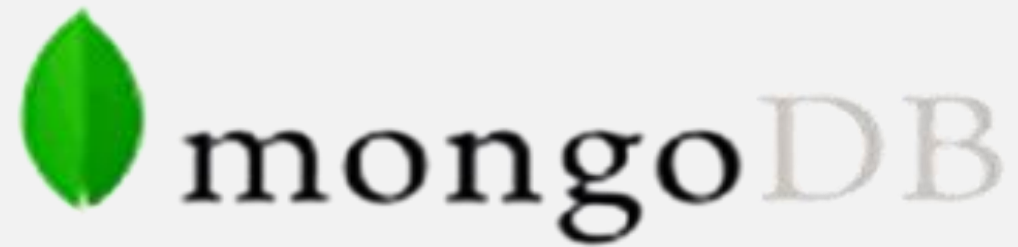


# MongoDB



## Introduction à NoSQL & MongoDB

comprendre les bases du NoSQL et découvrir MongoDB.



# Qu'est-ce que le NoSQL ?

---

**NoSQL** signifie “**Not Only SQL**” (pas seulement SQL).

Ce terme désigne une famille de bases de données **non relationnelles**, conçues pour gérer de grands volumes de données, souvent non structurées, avec plus de flexibilité que les bases traditionnelles (SQL).

# Pourquoi le NoSQL ?

---

## Pas de schéma fixe :

- Contrairement aux bases SQL (tables avec colonnes rigides), une base NoSQL permet d'ajouter des champs différents dans chaque enregistrement (document).
- Exemple : un document "utilisateur" peut avoir un champ *adresse* tandis qu'un autre ne l'a pas.

## Scalabilité horizontale :

- Les bases NoSQL s'adaptent facilement à la croissance en répartissant les données sur plusieurs serveurs (*sharding*).
- Parfait pour les applications qui grossissent très vite (réseaux sociaux, e-commerce, IoT).

## Performance élevée :

- Optimisées pour lire/écrire rapidement un grand volume de données.

## Formats variés :

- Données stockées sous forme **clé-valeur**, **colonnes**, **graphes**, ou **documents JSON** (comme MongoDB).

# Quand utiliser le NoSQL ?

- Applications avec **énormément de données** (Big Data).
- Quand les données sont **peu structurées ou évolutives**.
- Pour des systèmes qui doivent être **hautement disponibles et rapides**.
  - Exemple : Facebook, Netflix, Amazon utilisent NoSQL.



# Différences SQL vs NoSQL









---

## ***SQL - NoSQL***

<b>SQL (Relationnel)</b>	<b>NoSQL (Non relationnel)</b>
Schéma fixe	Schéma flexible
Tables, colonnes	Collections, documents
Relations fortes	Relations souples
Transactions complexes	Scalabilité facile

# Différences SQL vs NoSQL

## SQL - NoSQL

*RELATIONNEL  <b>SQL</b>	VS	<b>NoSQL</b>  *NON RELATIONNEL
Table	<b>STRUCTURE</b> 	Document
Statique	<b>SCHEMA</b> 	Dynamique
Oui	<b>ACID</b> 	Non
Oui	<b>JOIN</b> 	Non
Traitement multilignes avec jointures	<b>UTILISATION</b> 	Performant pour des données non structurées
Vertical Scaling	<b>SCALING</b> 	Horizontal scaling

# Types de bases NoSQL

---

**Clé – Valeur** : Redis

**Colonnes** : Cassandra

**Graphes** : Neo4j

**Documents** : MongoDB

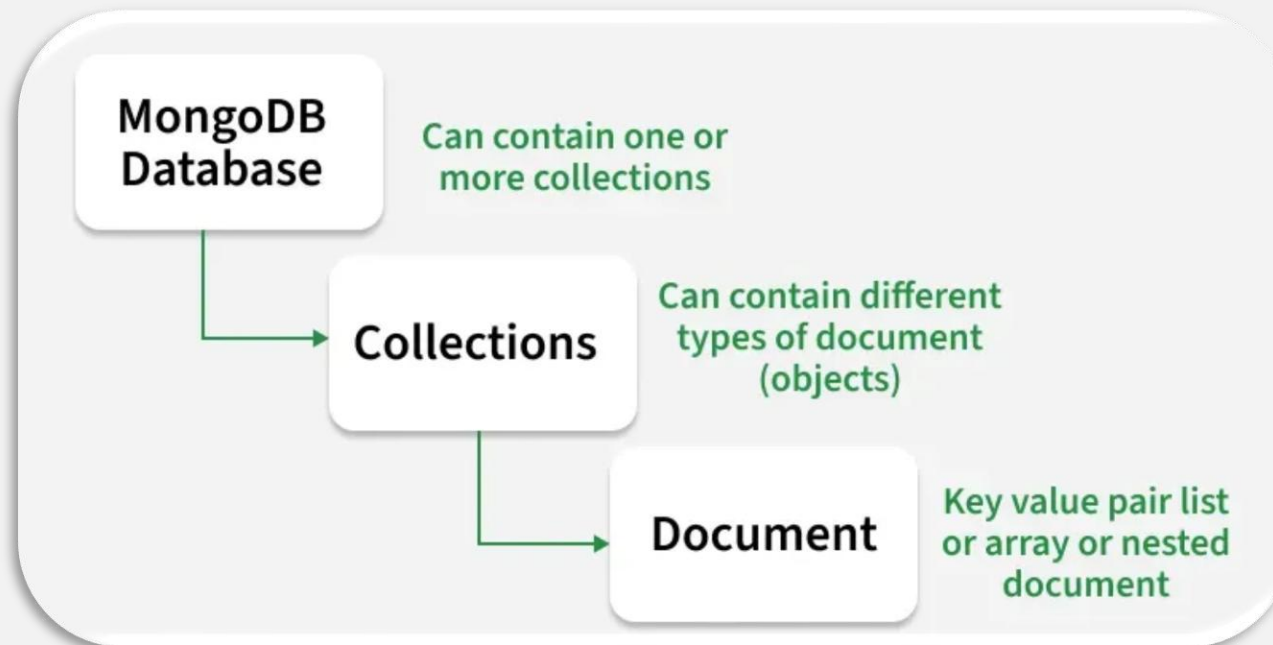




# MongoDB en bref

---

- Base de données **documentaire** (format BSON = JSON binaire).
- Données stockées dans :
  - **Database** → ensemble de collections
  - **Collection** → ensemble de documents
  - **Document** → enregistrement au format JSON



# Installation & Outils

---

- **MongoDB Community Server** (local)
  - version gratuite à installer sur son ordinateur, pour travailler en local.
- **MongoDB Atlas** (cloud, gratuit)
  - base de données hébergée dans le cloud, facile à déployer et gratuite jusqu'à un certain quota.
- **MongoDB Compass** (GUI pour requêtes)
  - interface graphique pour visualiser, créer et interroger les données sans passer par la ligne de commande.
- **Shell** : *`mongosh`*
  - terminal interactif pour exécuter des commandes MongoDB (remplace l'ancien mongo).



<https://www.mongodb.com/>

# Première base & collection

---

```
// Se connecter à MongoDB
mongosh

// Créer une base
use bibliotheque

// Créer une collection et insérer un document
db.livres.insertOne({
  titre: "L'Alchimiste",
  auteur: "Paulo Coelho",
  annee: 1988,
  pages: 208
})

// Lire les documents
db.livres.find()
```

# Exercices pratiques

---

1. Créer une base *bibliothèque*.
2. Créer une collection *livres*.
3. Insérer 5 livres (titre, auteur, année, pages).
4. Afficher tous les livres.
5. Afficher uniquement ceux publiés après 2010.

# CRUD

---

## CRUD et Opérateurs dans MongoDB

maîtriser les opérations **Créer, Lire, Mettre à jour, Supprimer** et utiliser les **opérateurs**.



create



read



update



delete

# Rappel CRUD

---

**C** : Create → insérer un document

**R** : Read → lire/rechercher un document

**U** : Update → modifier un document

**D** : Delete → supprimer un document



# CRUD - *CREATE*

---

## Insertion de documents

```
// Un seul document
db.etudiants.insertOne({
  nom: "Alice",
  age: 22,
  formation: "DWM",
  note: 16
})

// Plusieurs documents
db.etudiants.insertMany([
  { nom: "Bob", age: 24, formation: "DWM", note: 14 },
  { nom: "Sam", age: 21, formation: "CDA", note: 18 }
])
```

# CRUD - *READ*

---

## Lecture document

```
// Tous les documents
db.etudiants.find()

// Avec condition
db.etudiants.find({ formation: "DWM" })

// Avec projection (afficher seulement nom et note)
db.etudiants.find({}, { nom: 1, note: 1, _id: 0 })
```



# CRUD - *UPDATE*

---

Mise à jour du document

```
// Modifier un champ
db.etudiants.updateOne(
  { nom: "Alice" },
  { $set: { note: 18 } }
)

// Ajouter +1 à la note de tous les DWM
db.etudiants.updateMany(
  { formation: "DWM" },
  { $inc: { note: 1 } }
)
```

# CRUD - *DELETE*

---

## Suppression de document

```
// Supprimer un étudiant
db.etudiants.deleteOne({ nom: "Bob" })

// Supprimer plusieurs étudiants
db.etudiants.deleteMany({ note: { $lt: 10 } })
```

# Opérateurs de comparaison

---

Opérateurs de comparaison (comparaison de valeurs) :

**\$eq** : égal à

**\$ne** : différent de

**\$gt**, **\$gte** : supérieur (ou égal)

**\$lt**, **\$lte** : inférieur (ou égal)

**\$in** : valeur dans une liste

**\$nin** : valeur pas dans une liste

Exemple :

```
db.etudiants.find({ note: { $gte: 15 } })  
db.etudiants.find({ formation: { $in: ["DWM", "CDA"] } })
```

# Opérateurs de comparaison

---

Opérateurs de mise à jour (dans update, pas find)

Utilisés avec updateOne, updateMany

Opérateur	Rôle	Exemple
\$set	Modifie ou crée une clé	{ \$set: { pays: "France" } }
\$unset	Supprime une clé	{ \$unset: { age: "" } }
\$inc	Incrémente une valeur numérique	{ \$inc: { score: 1 } }
\$push	Ajoute un élément à un tableau	{ \$push: { tags: "js" } }
\$pull	Retire un élément d'un tableau	{ \$pull: { tags: "php" } }
\$addToSet	Ajoute un élément si non présent	{ \$addToSet: { tags: "node" } }

# Opérateurs de comparaison

---

Opérateurs logiques - Permettent de combiner plusieurs conditions.

Opérateur	Signification	Exemple
\$and	Toutes les conditions doivent être vraies	{ \$and: [{ age: { \$gte: 18 } }, { ville: "Marseille" }] }
\$or	Au moins une condition vraie	{ \$or: [{ age: { \$lt: 18 } }, { ville: "Marseille" }] }
\$nor	Aucune des conditions n'est vraie	{ \$nor: [{ ville: "Marseille" }, { ville: "Paris" }] }
\$not	Inverse le résultat d'une condition	{ age: { \$not: { \$gt: 18 } } }

# Opérateurs de comparaison

---

## Opérateurs d'éléments (structure du document)

Permettent de tester la présence ou le type d'un champ.

Opérateur	Signification	Exemple
\$exists	Vérifie si un champ existe	{ age: { \$exists: true } }
\$type	Vérifie le type de la valeur	{ age: { \$type: "number" } }

# Opérateurs de comparaison

## Opérateurs d'évaluation (conditions spéciales)

Permettent de filtrer selon des expressions régulières ou du code.

Opérateur	Signification	Exemple
<code>\$regex</code>	Recherche par motif (expression régulière)	<code>{ nom: { \$regex: "^L", \$options: "i" } }</code> → noms commençant par "A"
<code>\$expr</code>	Compare des champs dans un même document	<code>{ \$expr: { \$gt: ["\$revenu", "\$dépenses"] } }</code>
<code>\$mod</code>	Vérifie un reste de division	<code>{ age: { \$mod: [5, 0] } }</code> → multiples de 5
<code>\$text</code>	Recherche textuelle (index textuel requis)	<code>{ \$text: { \$search: "MongoDB" } }</code>

# Opérateurs de comparaison

---

## Opérateurs d'ensembles (listes et tableaux)

Permettent de comparer ou vérifier des éléments de tableaux.

Opérateur	Signification	Exemple
\$all	Tous les éléments doivent être présents	{ tags: { \$all: ["js", "node"] } }
\$elemMatch	Condition sur au moins un élément du tableau	{ notes: { \$elemMatch: { \$gt: 15, \$lt: 20 } } }
\$size	Taille exacte d'un tableau	{ tags: { \$size: 3 } }



# Exercices pratiques

---

1. Insérer 5 étudiants avec les champs (nom, âge, formation, note).
2. Afficher tous les étudiants en DWWM.
3. Afficher uniquement le nom et la note des étudiants.
4. Augmenter de +2 la note de tous les étudiants en CDA.
5. Supprimer les étudiants avec une note  $< 8$ .

# RELATIONS

---

## Relations & Agrégations

- Comprendre comment gérer les **relations entre collections** (équivalent des jointures en SQL).
- Maîtriser la différence entre **documents embarqués (embedded)** et **références**.
- Savoir utiliser le **pipeline d'agrégation** de MongoDB pour analyser et transformer les données.
- Découvrir la notion **d'index** pour améliorer les performances.



# Relations entre collections

---

## Concepts

- **Embedded documents** (documents imbriqués dans un autre).  
Exemple : un client avec directement son adresse dans le même document.
- **Références** (stockage d'un ObjectId vers une autre collection).  
Exemple : une commande qui référence un client par son id.

Comparaison :

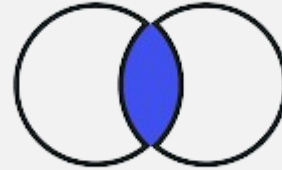
Embedded = plus rapide mais plus lourd,

Références = plus flexible et proche du SQL.

# Jointure avec *\$lookup*

Equivalent du JOIN en SQL.

Syntaxe basique :



```
db.commandes.aggregate([
  {
    $lookup: {
      from: "clients",           // collection cible
      localField: "clientId",    // champ dans commandes
      foreignField: "_id",       // champ dans clients
      as: "client_info"         // alias du résultat
    }
  }
])
```

# Le pipeline d'agrégation

---

MongoDB utilise un système de **pipeline** (enchaînement d'étapes).

## Opérateurs courants :

- **\$match** → filtrer les documents.
- **\$group** → regrouper et appliquer des fonctions d'agrégation (\$sum, \$avg, \$max).
- **\$sort** → trier.
- **\$limit** → limiter le nombre de résultats.
- **\$project** → sélectionner ou reformater des champs.

Exemple :

```
db.commandes.aggregate([
  { $match: { statut: "livrée" } },
  { $group: { _id: "$clientId", totalCommandes: { $sum: 1 } } },
  { $sort: { totalCommandes: -1 } }
])
```

# Notion d'index

---

Par défaut MongoDB indexe `_id`.

On peut créer des index sur d'autres champs pour accélérer les recherches :

```
db.clients.createIndex({ email: 1 })
```

# Exercices pratiques

---

## Exercice 1 — Embedded documents

Créer une collection `clients` avec un champ `adresse` imbriqué (rue, ville, codePostal).

Insérer **3** clients et les afficher avec `find()`.

# Exercices pratiques

---

## Exercice 2 — Références et *\$lookup*

1. Créer une collection *commandes* avec *clientId* qui référence un client.
2. Ajouter plusieurs commandes liées aux clients.
3. Faire un *\$lookup* pour afficher les infos clients dans chaque commande.



# Exercices pratiques

---

## Exercice 3 — Agrégations

1. Compter le nombre de commandes par client.
2. Calculer la dépense totale de chaque client (avec `$group` et `$sum`).
3. Lister les 3 clients qui ont dépensé le plus.

# Exercices pratiques

---

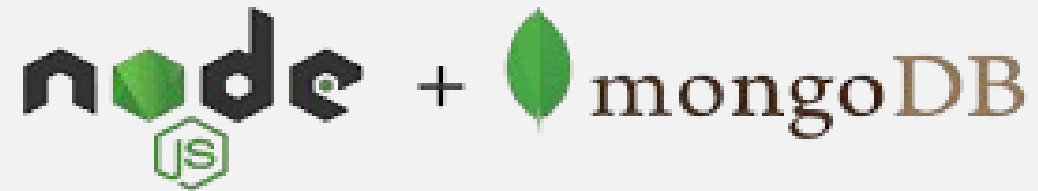
## Exercice 4 — Index

1. Créer un index sur le champ ville de la collection clients.
2. Tester une requête de recherche **avec et sans index**

SUITE TP — E-COMMERCE

# NODE.JS & MONGODB

---



- Savoir connecter une application Node.js à MongoDB.
- Découvrir **Mongoose** (modèles, schémas, validation).
- Créer une **API REST simple** avec Express + MongoDB.
- Implémenter un CRUD complet sur une ressource.
- Tester l'API avec Postman / Insomnia.

# NODE.JS & MONGODB

---

## 1. Rappel & Préparation

### Contenu

- Notion de **driver MongoDB natif** vs **ODM Mongoose**.
- Installation :

```
npm init -y  
npm install express mongoose nodemon cors
```

- Structure projet :

```
project/  
├── models/  
│   └── User.js  
├── routes/  
│   └── userRoutes.js  
└── index.js
```

# NODE.JS & MONGODB

---

## 2. Connexion à MongoDB avec Mongoose

Code (*index.js*)

```
import express from "express";
import { connect } from "mongoose";
import cors from "cors";
import userRoutes from "../routes/userRoutes.js";

const app = express();
app.use(express.json());

app.use("/users", userRoutes);
app.use(cors());

connect("mongodb://127.0.0.1:27017/dwwm_api")
  .then(() => console.log("Connecté à MongoDB"))
  .catch(err => console.error("Erreur MongoDB :", err));

app.get("/", (req, res) => res.send("API MongoDB en Node.js"));
app.listen(3000, () => console.log("Serveur lancé sur http://localhost:3000"));
```

# NODE.JS & MONGODB

---

## 3. Créer un modèle avec Mongoose

Exemple *models/User.js*

```
import { Schema, model } from "mongoose";

const userSchema = new Schema({
  nom: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  motdepasse: { type: String, required: true },
  role: { type: String, enum: ["user", "admin"], default: "user" },
  dateCreation: { type: Date, default: Date.now },
});

export default model("User", userSchema);
```

# NODE.JS & MONGODB

---

## 4. Créer des routes CRUD - **CREATE**

Exemple routes/userRoutes.js

```
import { Router } from "express";
const router = Router();
import User from "../models/User.js";

// CREATE
router.post("/", async (req, res) => {
  try {
    const user = new User(req.body);
    const savedUser = await user.save();
    res.status(201).json(savedUser);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```

# NODE.JS & MONGODB

---

## 4. Créer des routes CRUD - **READ**

Exemple routes/userRoutes.js

```
// READ - tous les utilisateurs
router.get("/", async (req, res) => {
  const users = await User.find();
  res.json(users);
});

// READ - un utilisateur
router.get("/:id", async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) return res.status(404).json({ error: "Utilisateur non trouvé" });
    res.json(user);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```



# NODE.JS & MONGODB

---

## 4. Créer des routes CRUD - **UPDATE**

Exemple routes/userRoutes.js

```
// UPDATE
router.put("/:id", async (req, res) => {
  try {
    const user = await User.findByIdAndUpdate(req.params.id, req.body, {
      new: true,
    });
    if (!user) return res.status(404).json({ error: "Utilisateur non trouvé" });
    res.json(user);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```

# NODE.JS & MONGODB

---

## 4. Créer des routes CRUD - **DELETE**

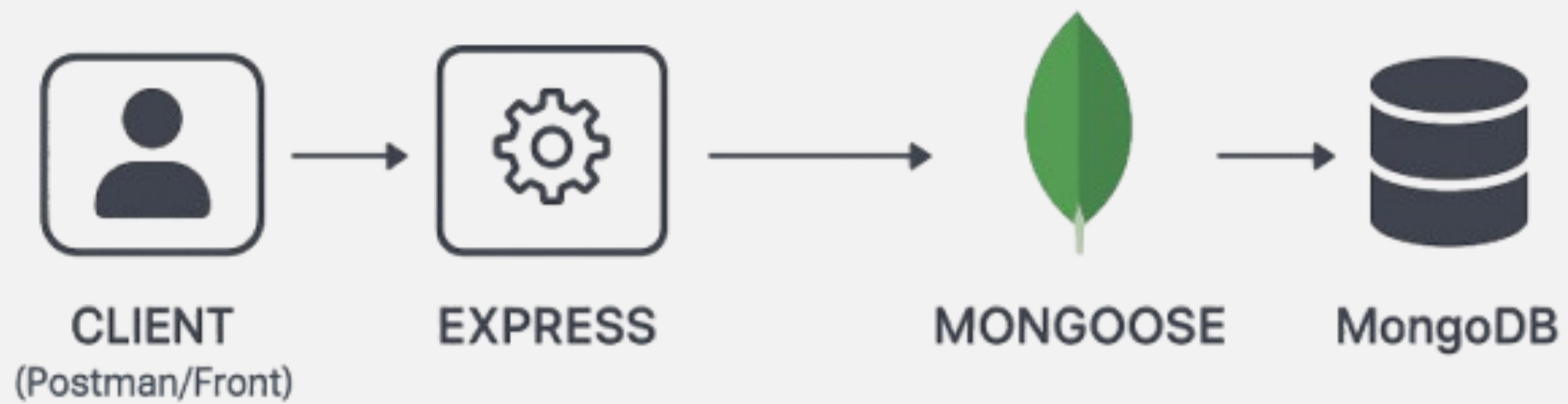
Exemple routes/userRoutes.js

```
// DELETE
router.delete("/:id", async (req, res) => {
  try {
    const user = await User.findByIdAndDelete(req.params.id);
    if (!user) return res.status(404).json({ error: "Utilisateur non trouvé" });
    res.json({ message: "Utilisateur supprimé" });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

export default router;
```

# NODE.JS & MONGODB

---



# NODE.JS & MONGODB

---

## 5. Tester avec Postman

- POST /users → créer un utilisateur
- GET /users → récupérer tous les utilisateurs
- GET /users/:id → récupérer un utilisateur
- PUT /users/:id → modifier un utilisateur
- DELETE /users/:id → supprimer un utilisateur

# Exercices

---

## Exercices pratiques

### 1. Connexion

- Créer la base `dwwwm_api`.
- Lancer le serveur et vérifier la connexion.

### 2. Modèle User

- Ajouter un champ `age` (optionnel).
- Ajouter une validation : `age >= 18`.

### 3. CRUD Users

- Créer 3 utilisateurs via `POST /users`.
- Lister tous les utilisateurs.
- Modifier un utilisateur (changer le rôle).
- Supprimer un utilisateur.

### 4. Extension Produits

- Créer un modèle `Product` avec (nom, prix, stock, actif).
- Implémenter les routes CRUD `/products`.

# TRAVAUX PRATIQUES

---

**TP**