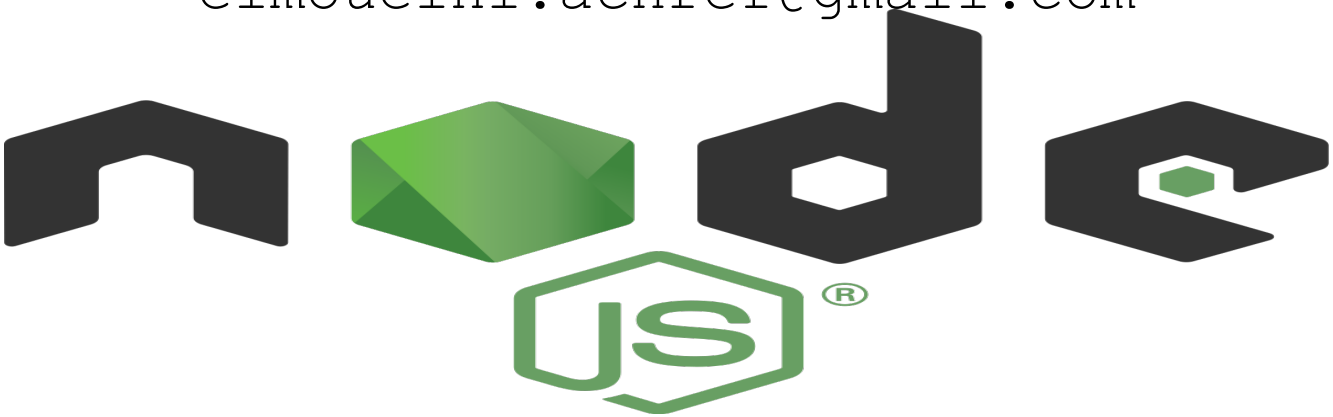
Node.js : connexion à **MongoDB**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com



Plan

Plan

Plan

7. CRUD avec `mongoose`
   - `save()`
   - `create()`
   - `insertMany()`
   - `findOne()`
   - `find()`
   - `find({ ... })`
   - `findById()`
   - `updateOne()`
   - `updateMany()`
   - `deleteOne()`
   - `deleteMany()`

Plan

Node.js

Objectif

Se connecter à une base de données **MongoDB** depuis une application **Node.js**

Node.js

Objectif

Se connecter à une base de données **MongoDB** depuis une application **Node.js**

Au moins deux packages **NPM** disponibles

- `mongodb`
- `mongoose`

Node.js

Package **mongodb**

- Pilote officiel **MongoDB** pour **Node.js**
- **API** de bas niveau
- Permettant d'interagir avec une base de données **MongoDB** en utilisant le langage **JavaScript**
- Prenant en charge les **callbacks** et les **promesses**

Node.js

Package **mongoose**

- Bibliothèque open-source créé par **Aaron Heckmann**
- Hébergé sur **GitHub**
- Offrant une couche d'abstraction qui simplifie l'interaction avec **MongoDB**
- Fournissant un schéma de données, des fonctions de validation, des hooks...

Node.js

Étapes

- Création d'un nouveau projet **Node.js**
- Installation du pilote **mongodb**
- Importation des modules **mongodb** installés
- Utilisation

Node.js

Démarche

- Créez un répertoire `cours-mongo` dans votre espace de travail
- Lancez **VSC** et allez dans `File > Open Folder...` et choisissez `cours-mongo`
- Dans `cours-mongo`, créez un fichier `main.js`

Node.js

Initiation d'un projet `Node.js`

```
npm init -y
```

Node.js

Initiation d'un projet Node.js

```
npm init -y
```

Lancement du projet

```
nodemon .
```

Node.js

Installation de mongodb

```
npm install mongodb
```

Node.js

Installation de mongodb

```
npm install mongodb
```

Importation du module installé (dans main.js)

```
const { MongoClient } = require('mongodb');
```

Node.js

Utilisation

- Préparation et établissement de la connexion avec la base de données
- Exécution des requêtes **NoSQL**
- Clôture de la connexion

Node.js

Commençons par préparer la chaîne de connexion

```
const uri = "mongodb://127.0.0.1";
```

Node.js

Commençons par préparer la chaîne de connexion

```
const uri = "mongodb://127.0.0.1";
```

Construisons ensuite un objet client MongoDB

```
const client = new MongoClient(uri);
```

Node.js

Commençons par préparer la chaîne de connexion

```
const uri = "mongodb://127.0.0.1";
```

Construisons ensuite un objet client MongoDB

```
const client = new MongoClient(uri);
```


Définissons enfin une fonction dans laquelle on établit la connexion et on exécute les requêtes

```
async function run() {

}
```

Node.js

Dans la fonction `run`, on établit la connexion avec la base de données et on la ferme après utilisation

```
async function run() {
    try {
        await client.connect();

    } finally {
        await client.close();
    }
}
```

Node.js

Et pour tout tester, il faut exécuter la promesse

```
run()
    .catch(console.error);
```

Node.js

Créons une instance de la base de données passée en paramètre

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');

    } finally {
        await client.close();
    }
}
```

Node.js

Récupérons une référence de la collection passée en paramètre, si elle n'existe pas alors elle sera créée

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');

    } finally {
        await client.close();
    }
}
```

Node.js

Pour récupérer le premier document de toute la collection

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const first = await personnes.findOne();
        console.log(first);

    } finally {
        await client.close();
    }
}
```

Node.js

Pour récupérer le premier document de toute la collection selon un filtre

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const first = await personnes.findOne({ prenom: 'John' });
        console.log(first);

    } finally {
        await client.close();
    }
}
```

Node.js

Pour récupérer tous les documents de la collection

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const documents = await personnes.find().toArray();

        for (const doc of documents) {
            console.log(doc);
        }

    } finally {
        await client.close();
    }
}
```

Node.js

La méthode `find()` ne retourne pas un tableau, elle retourne un curseur

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const list = personnes.find()

        while (await list.hasNext()) {
            console.log(await list.next());
        }
    } finally {
        await client.close();
    }
}
```

Node.js

Une deuxième version avec `for ... of`

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const list = personnes.find()

        for await (const item of list) {
            console.log(item);
        }

    } finally {
        await client.close();
    }
}
```

Node.js

Pour récupérer tous les documents respectant certains critères

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const documents = await personnes.find({ nom: { $ne: "Wick" } }).toArray();

        for (const doc of documents) {
            console.log(doc);
        }

    } finally {
        await client.close();
    }
}
```

Node.js

Pour insérer un nouveau document

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const doc = { nom: "Dalton", prenom: "Jack" };
        const result = await personnes.insertOne(doc);
        console.log(`Document inséré avec _id = ${result.insertedId}`);
    } finally {
        await client.close();
    }
}
```

Node.js

Pour insérer plusieurs documents

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');
        const docs = [
            { nom: 'Hood', prenom: 'Lucas' },
            { nom: 'Carlos', prenom: 'Roberto' },
            { prenom: 'Peter', nom: 'Schmeikel' }
        ]
        const result = await personnes.insertMany(docs);
        let ids = result.insertedIds;
        for (const id of Object.values(ids)) {
            console.log(id);
        }
    } finally {
        await client.close();
    }
}
```

Node.js

Pour modifier un document

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');

        const filter = { prenom: "John" };
        const newValues = {
            $set: {
                nom: "Travolta",
            },
        };

        const result = await personnes.updateOne(filter, newValues);
        console.log(result);
    } finally {
        await client.close();
    }
}
run().catch(console.error);
```

Node.js

Exécution de requête : suppression de données

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');

        const filter = {
            nom: {
              $eq: 'Wick',
            }
          };

        const result = await personnes.deleteOne(filter);
        console.log(result);
    } finally {
        await client.close();
    }
}
```

Node.js

Exécution de requête : suppression de données

```
async function run() {
    try {
        await client.connect();
        const db = client.db('cours');
        const personnes = db.collection('personnes');

        const filter = {
            nom: {
              $eq: 'Dalton',
            }
          };

        const result = await personnes.deleteMany(filter);
        console.log(result);
    } finally {
        await client.close();
    }
}
```

Node.js

Étapes

- Création d'un nouveau projet **Node.js**
- Installation du pilote **mongoose**
- Importation des modules **mongoose** installés
- Utilisation

Node.js

Démarche

- Créez un répertoire `cours-mongoose` dans votre espace de travail
- Lancez **VSC** et allez dans `File > Open Folder...` et choisissez `cours-mongoose`
- Dans `cours-mongoose`, créez un fichier `main.js`

Node.js

Initiation d'un projet `Node.js`

```
npm init -y
```

Node.js

Initiation d'un projet Node.js

```
npm init -y
```

Lancement du projet

```
nodemon .
```

Node.js

Installation de mongoose

```
npm install mongoose
```

Node.js

Installation de mongoose

```
npm install mongoose
```

Importation du module installé (dans main.js)

```
const mongoose = require('mongoose');
```

Node.js

Utilisation

- Préparation et établissement de la connexion avec la base de données
- Exécution des requêtes **NoSQL**
- Clôture de la connexion

Node.js

Commençons par préparer la chaîne de connexion

```
const uri = "mongodb://127.0.0.1:27017/cours";
```

Node.js

Commençons par préparer la chaîne de connexion

```
const uri = "mongodb://127.0.0.1:27017/cours";
```

Définissons enfin une fonction dans laquelle on établit la connexion et on exécute les requêtes

```
async function run() {

}
```

Node.js

Commençons par préparer la chaîne de connexion

```
const uri = "mongodb://127.0.0.1:27017/cours";
```

Définissons enfin une fonction dans laquelle on établit la connexion et on exécute les requêtes

```
async function run() {

}
```


Établissons la connexion avec la base de données

```
async function run() {
    await mongoose.connect(uri);
}
```

Node.js

N'oublions pas d'appeler la fonction `run`

```
run()
    .catch(console.error);
```

Node.js

Préparons le schéma de la collection

```
async function run() {
    await mongoose.connect(uri);

    const personneSchema = new mongoose.Schema({
        nom: String,
        prenom: String
    });
}
```

Node.js

Quelques autres types

- ObjectId
- Number
- Boolean
- Date
- UUID
- Array
- ...

Node.js

Ensuite le modèle

```
async function run() {
    await mongoose.connect(uri);

    const personneSchema = new mongoose.Schema({
        nom: String,
        prenom: String
    });

    const Personne = mongoose.model('Personne', personneSchema);
}
```

Node.js

On peut aussi fusionner les deux instructions précédentes

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });
}
```

Node.js

Remarques

- Par défaut, la collection portera le nom du modèle en minuscule et au pluriel.
- Pour un nom de collection différent, on peut le spécifier comme troisième paramètre de la méthode `model`.

Node.js

Pour insérer un nouveau document, utilisons la méthode `save`

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personne = new Personne({ nom: "Linus", prenom: 'Benjamin' })

    personne.save()
        .then(res => console.log(`insertion réussie, id : ${res._id}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Node.js

Remarques

- À chaque insertion d'un nouveau document, **Mongoose** ajoute un champ interne `__v`.
- Ce champ est utilisé pour le contrôle de la version des documents.
- À chaque mise à jour, **Mongoose** incrémente automatiquement la valeur de `__v`.
- **Mongoose** pourrait l'utiliser pour détecter les conflits de mise à jour concurrente.

Ajoutons un document avec un champ ne figurant pas dans le schéma

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personne = new Personne({ nom: "Linus", prenom: 'Benjamin',
      age: 57 })

    personne.save()
        .then(res => console.log(`insertion réussie, id : ${res._id}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Ajoutons un document avec un champ ne figurant pas dans le schéma

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personne = new Personne({ nom: "Linus", prenom: 'Benjamin',
       age: 57 })

    personne.save()
        .then(res => console.log(`insertion réussie, id : ${res._id}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Vérifiez que l'`age` n'a pas été inséré.

Pour autoriser l'insertion des champs ne figurant pas dans le schéma, on ajoute la propriété , `{ strict: false }`

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', new mongoose.Schema({
        nom: String,
        prenom: String
    }, { strict: false }));

    const personne = new Personne({ nom: "Linus", prenom: 'Benjamin',
       age: 57 })

    personne.save()
        .then(res => console.log(`insertion réussie, id : ${res._id}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Pour autoriser l'insertion des champs ne figurant pas dans le schéma, on ajoute la propriété , `{ strict: false }`

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', new mongoose.Schema({
        nom: String,
        prenom: String
    }, { strict: false }));

    const personne = new Personne({ nom: "Linus", prenom: 'Benjamin',
       age: 57 })

    personne.save()
        .then(res => console.log(`insertion réussie, id : ${res._id}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Vérifiez que l'`age` a été inséré.

Node.js

On peut aussi spécifier les valeurs au moment de la création de l'objet

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    Personne.create({ nom: "Shephard", prenom: 'Jack', age: 45 })
        .then(res => console.log(`insertion réussie, id : ${res._id}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Node.js

Pour insérer plusieurs documents

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    Personne.insertMany([
        { nom: "Ford" },
        { prenom: 'John', nom: 'Lock' }
    ])
        .then(res => console.log(`insertion réussie, id : ${res}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Node.js

Ou avec `create`

```
async function run() {

    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    Personne.create([
        { prenom: "James", age: 37 },
        { prenom: 'John', nom: 'Abruzzi' }
    ])
        .then(res => console.log(`insertion réussie, id : ${res}`))
        .catch(err => `Problème d'insertion : ${err}`)
}
```

Node.js

Pour récupérer le premier document de toute la collection

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personne = await Personne.findOne();
    console.log(personne);

}
```

Node.js

Pour récupérer le premier document de toute la collection selon un filtre

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personne = await Personne.findOne({ prenom: 'John' });
    console.log(personne);
}
```

Node.js

Pour récupérer tous les documents de la collection

```
async function run() {
    await mongoose.connect(uri);
    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personnes = await Personne.find();
    for (const item of personnes) {
        console.log(item);
    }
}
```

Node.js

Pour récupérer tous les documents de la collection

```
async function run() {
    await mongoose.connect(uri);
    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personnes = await Personne.find({ nom: 'Wick' });
    for (const item of personnes) {
        console.log(item);
    }
}
```

Pour appliquer un filtre sur les documents, on peut aussi utiliser les constructeurs de `Query` (`where`, `equals`...)

```
async function run() {
    await mongoose.connect(uri);
    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personnes = Personne.find()
        .where('nom')
        .equals('Carlos')
        .exec();

    personnes
        .then(res => console.log(res))
        .catch(console.err)
}
```

Pour appliquer un filtre sur les documents, on peut aussi utiliser les constructeurs de `Query` (`where`, `equals`...)

```
async function run() {
    await mongoose.connect(uri);
    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personnes = Personne.find()
        .where('nom')
        .equals('Carlos')
        .exec();

    personnes
        .then(res => console.log(res))
        .catch(console.err)
}
```


Explication

- `where` : pour cibler le champ sur lequel la condition doit s'appliquer
- `equals` : pour vérifier l'égalité de valeur avec le champ indiqué dans `where`
- `exec` : pour créer la promesse

Node.js

Il est possible d'avoir plusieurs `where` [toutes les conditions doivent être vraies (AND)]

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personnes = Personne.find()
        .exists('age')
        .where('age')
        .gt(30)
        .where('nom')
        .ne('Wick')
        .exec()

    personnes
        .then(res => console.log(res))
        .catch(console.err)
}
```

Node.js

Explication

- Pour comparer les chaînes de caractères : `equals` et `ne`
- Pour comparer les nombres : `gt`, `lt`, `gte` et `lte`
- Pour tester l'inclusion (dans une liste) : `in` et `nin`
- Pour les opérateurs logiques : `or([c1, c2])` et `and([c1, c2])`
- Pour la recherche textuelle : `regex(/expression/)`
- Pour trier : `sort({ champ: 1 })` (1 pour croissant, -1 pour décroissant)
- Pour déselectionner quelques documents : `limit()` et `skip`
- Pour sélectionner quelques champs (projection) : `select`
- ...

Node.js

Pour plus de détails sur les Query

https://mongoosejs.com/docs/api/query.html

Node.js

Exercice

Afficher la liste des personnes dont le nom commence par `W` ou le prénom se termine par `n`

Node.js

Solution

```
async function run() {
    await mongoose.connect(stringConnection)

    const Personne = mongoose.model('Personne', new mongoose.Schema({
        nom: String,
        prenom: String
    }, { strict: false }))

    try {
        const personnes = await Personne.find()
            .or([
                { nom: { $regex: /^W/i } },
                { prenom: { $regex: /n$/, $options: 'i' } }
            ])
            .exec()
        console.log(personnes);
    } catch {
        console.log('Problème de récupération de données');
    }

}
```

Node.js

Remarques

- Les méthodes de recherche selon l'identifiant s'attend à une chaîne de caractère d'un `ObjectId`
- Si l'`_id` est de type `Number`, il faut le spécifier dans le schéma
- **Mongoose** ne supporte pas la mixité de type pour `_id`

Node.js

Pour récupérer un document selon la valeur de son identifiant

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const personne = await Personne.findById(3);
    console.log(personne);

}
```

Node.js

Pour modifier un document

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const resultat = await Personne.updateOne(
        { nom: 'Wick' },
        { nom: 'Travolta', age: 65 }
    );
    console.log(resultat);

}
```

Node.js

Pour modifier plusieurs documents

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const resultat = await Personne.updateMany(
        { nom: 'Wick' },
        { nom: 'Travolta', age: 65 }
    );
    console.log(resultat);

}
```

Node.js

Pour supprimer un document

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const resultat = await Personne.deleteOne({ nom: 'Carlos' });
    console.log(resultat);
}
```

Node.js

Pour supprimer plusieurs documents

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: String,
        prenom: String
    });

    const resultat = await Personne.deleteMany({ nom: 'Carlos' });
    console.log(resultat);
}
```

Node.js

Remarques

- Avant insertion dans la base de données, il faut valider les données.
- Les règles de validation peuvent être définies dans le schéma

Node.js

Ici, nous déclarons `nom` comme étant un champ obligatoire de type chaîne de caractère et `prenom` optionnel de type chaîne de caractère aussi

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: {
            type: String,
            required: true
        },
        prenom: {
            type: String,
            required: false
        }
    });
}
```

Node.js

Autres validateurs selon le type

- **String** : `required`, `minlength`, `maxlength`, `enum`, `match` (pour les expressions régulières).
- **Number** : `required`, `min`, `max`.
- **Date** : `required`, `min`, `max`.

Node.js

Pour insérer uniquement si les valeurs sont valides, on peut appeler la méthode `validate`

```
async function run() {
    await mongoose.connect(uri);

    const Personne = mongoose.model('Personne', {
        nom: {
            type: String,
            required: true
        },
        prenom: {
            type: String,
            required: false
        }
    });

    const personne = new Personne({ prenom: 'Benjamin' })

    personne.validate()
        .then(() => {
            personne.save()
                .then((res) => console.log(res))
                .catch(err => console.log(`Problème de validation : ${err} `))
        })
        .catch(err => console.log(`Problème d'insertion : ${err}`))
}
```

Node.js

On peut aussi utiliser `try ... catch` et `await` pour éviter l'imbrication des promesses

```
async function run() {
    try {
        await mongoose.connect(uri);

        const Personne = mongoose.model('Personne', {
            nom: {
                type: String,
                required: true
            },
            prenom: {
                type: String,
                required: false
            }
        });

        const personne = new Personne({ prenom: 'Benjamin' });

        await personne.validate();
        const res = await personne.save();

        console.log(res);
    } catch (err) {
        console.error(`Problème d'insertion ou de validation : ${err}`);
    }
}
```

Node.js

Même exemple avec `create`

```
async function run() {
    try {
        await mongoose.connect(uri);

        const Personne = mongoose.model('Personne', {
            nom: {
                type: String,
                required: true
            },
            prenom: {
                type: String,
                required: false
            }
        });

        const personne = new Personne({ prenom: 'Benjamin' });

        await personne.validate();
        const res = await Personne.create(personne);

        console.log(res);
    } catch (err) {
        console.error(`Problème d'insertion ou de validation : ${err}`);
    }
}
```

Node.js

Ou

```
async function run() {
    try {
        await mongoose.connect(uri);

        const Personne = mongoose.model('Personne', {
            nom: {
                type: String,
                required: true
            },
            prenom: {
                type: String,
                required: false
            }
        });

        const personne = new Personne({ prenom: 'Benjamin' });

        await Personne.validate(personne);
        const res = await Personne.create(personne);

        console.log(res);
    } catch (err) {
        console.error(`Problème d'insertion ou de validation : ${err}`);
    }
}
```

Node.js

Remarques

- Les champs qui ne figurent pas dans le schéma seront insérés si la propriété `{ strict: false }` est présente.
- Les champs figurant dans le schéma doivent être valides avant insertion même si la propriété `{ strict: false }` est présente.

Node.js

Pour définir un validateur personnalisé, on utilise la clé `validate`

```
const Personne = mongoose.model('Personne', {
    nom: {
        type: String,
        required: true
    },
    prenom: {
        type: String,
        required: false
    },
    email: {
        type: String,
        validate: {
            validator: (value) => /\S+@\S+\.\S+/.test(value),
            message: "Email n'est pas valide"
        }
    }
});
```

Node.js

Pour définir un validateur personnalisé, on utilise la clé `validate`

```
const Personne = mongoose.model('Personne', {
    nom: {
        type: String,
        required: true
    },
    prenom: {
        type: String,
        required: false
    },
    email: {
        type: String,
        validate: {
            validator: (value) => /\S+@\S+\.\S+/.test(value),
            message: "Email n'est pas valide"
        }
    }
});
```

\S

Un caractère différent d'un whitespace (espace, retour à la ligne, tabulation...)

Node.js

Pour définir un message d'erreur personnalisé

```
const Personne = mongoose.model('Personne', {
    nom: {
        type: String,
        required: [true, 'Le champ {PATH} est obligatoire']
    },
    prenom: {
        type: String,
        required: false
    },
    email: {
        type: String,
        validate: {
            validator: (value) => /\S+@\S+\.\S+/.test(value),
            message: "Email n'est pas valide"
        }
    }
});
```

Node.js

Pour intégrer la valeur non-valide dans le message d'erreur, on utilise {VALUE}

```
const Personne = mongoose.model('Personne', {
    nom: {
        type: String,
        required: [true, 'Le champ {PATH} est obligatoire']
    },
    prenom: {
        type: String,
        required: false
    },
    email: {
        type: String,
        validate: {
            validator: (value) => /\S+@\S+\.\S+/.test(value),
            message: "{VALUE} n'est pas valide"
        }
    }
});
```