TP 1 : Installation et Manipulation de la BD NoSQL « MongoDB »

Filière : SUD, INE2

Encadré par : Pr. Dounia ZAIDOUNI

Les objectifs du TP : « Installation et Manipulation de MongoDB » sont les suivants :

- Installation et configuration de MongoDB sur une VM Ubuntu.

- Examination des fonctionnalités de requêtage de MongoDB :
  - Restauration d'un fichier « .bson » ,
  - Gestion des indexes,
  - Recherche et tri des documents ,
  - Insertion des documents,
  - Suppression des documents,
  - Mise à jour des documents.

- Implémentation d'une application avec Node.js et MongoDB et réalisation des opérations CRUD :
  - Create,
  - Read,
  - Update,
  - Delete.

1) Installation et configuration de MongoDB :

Installation et configuration sur une VM Ubuntu :

Pour installer MongoDB, deux types de paquets sont disponibles: le paquet fourni par la communauté ubuntu et le paquet fourni par la communauté mongodb. Le deuxième comporte la version la plus récente.

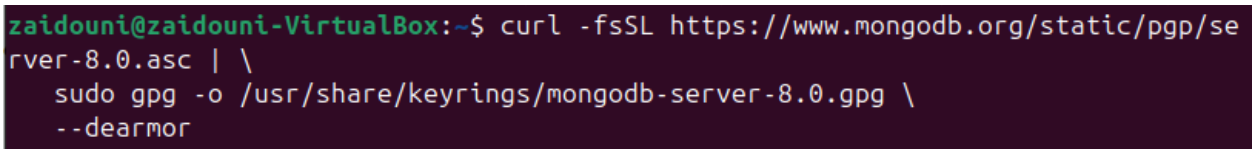
Dans ce TP, nous allons installer le paquet fourni par la communauté ubuntu, pour cela, suivez les étapes suivantes :

Étape 1 - Configuration du « apt Repository » :

Tout d'abord, importez la clé GPK pour le « MongoDB apt Repository » sur votre système à l'aide des commandes suivantes (faites attention lors du copier/coller des commandes, il faut enlever les espaces ajoutés).

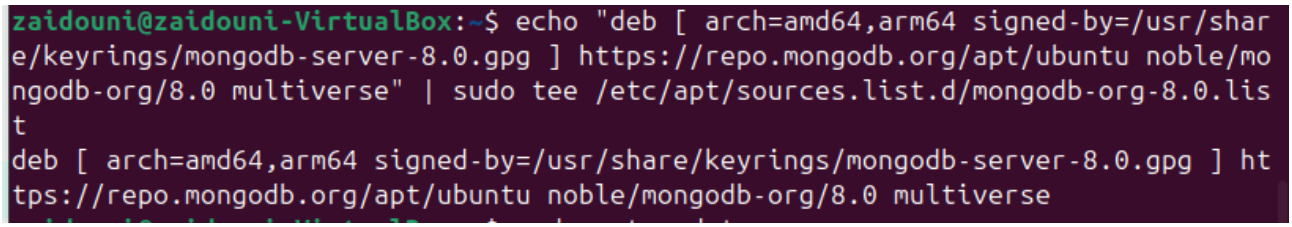
```
$ sudo apt-get install gnupg curl
```

```
$ curl -fsSL https://www.mongodb.org/static/pgp/server-8.0.asc |\
sudo gpg -o /usr/share/keyrings/mongodb-server-8.0.gpg --dearmor
```



Ensuite, créez le fichier liste **/etc/apt/sources.list.d/mongodb-org-8.0.list** pour ubuntu Noble, pour cela tapez :

```
$ echo "deb [ arch=amd64,arm64
signed-by=/usr/share/keyrings/mongodb-server-8.0.gpg ]
https://repo.mongodb.org/apt/ubuntu noble/mongodb-org/8.0
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org 8.0.list
```




```
$ sudo apt update
```

Étape 2 – Installation de MongoDB sur une VM Ubuntu

Utilisez la commande suivante pour installer MongoDB sur votre VM. Il installera également tous les packages dépendants requis pour MongoDB.

```
$ sudo apt-get install -y mongodb-org
```

Affichage de la version de MongoDB :

Pour Afficher, la version de MongoDB que vous avez installé, tapez :

```
$ mongod -version
```

```
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$ mongod -version
db version v8.0.15
Build Info: {
    "version": "8.0.15",
    "gitVersion": "f79b970f08f60c41491003cd55a3dd459a279c39",
    "openSSLVersion": "OpenSSL 3.0.13 30 Jan 2024",
    "modules": [],
    "allocator": "tcmalloc-google",
    "environment": {
        "distmod": "ubuntu2404",
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$
```

Gestion des services Mongod :

1- Activation et démarrage des services MongoDB :

```
$ sudo systemctl enable mongod
$ sudo systemctl start mongod
```

2- Affichage du statut de mongod :

```
$ sudo systemctl status mongod
```

```
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$ sudo systemctl status mongod
● mongod.service - MongoDB Database Server
     Loaded: loaded (/usr/lib/systemd/system/mongod.service; enabled; preset: enabled)
     Active: active (running) since Mon 2025-10-20 11:45:11 +01; 4s ago
       Docs: https://docs.mongodb.org/manual
   Main PID: 12139 (mongod)
     Memory: 89.3M (peak: 89.9M)
        CPU: 397ms
     CGroup: /system.slice/mongod.service
             └─12139 /usr/bin/mongod --config /etc/mongod.conf
```

3- Arrêt et redémarrage des services MongoDB :

Utilisez les commandes suivantes pour arrêter ou redémarrer le service MongoDB:

```
$ sudo systemctl stop mongod
$ sudo systemctl restart mongod
```

4- Test de la configuration :

Connectez MongoDB à l'aide de la ligne de commande et exécutez certaines commandes de test pour vérifier le bon fonctionnement. Tapez dans le terminal la commande : `$ mongosh`

Puis tapez les commandes suivantes :

```
> use mydb;
> db.test.insertOne( { zaidouni: 100 } )
> db.test.find()
```

2) Examination des fonctionnalités de requêtage de MongoDB

Restauration d'un fichier « .bson » :

Dans ce TP, nous allons utiliser les données stockées dans le fichier « **moviedetails.bson** » relatives aux informations sur plusieurs films. Récupérez ce fichier à partir de moodle, et déposez le dans /home/user/Documents.
Ensuite, tapez la commande suivante pour restaurer ces données dans la BD mongoDB « cinema » :

```
zaidouni@zaidouni:~/Documents$
mongorestore -d cinema -c films movieDetails.bson
```

```
zaidouni@zaidouni-VirtualBox:~/Documents$ mongorestore -d cinema -c films movieD
etails.bson
2024-12-11T11:57:06.184+0100    checking for collection data in movieDetails.bso
n
2024-12-11T11:57:06.238+0100    restoring cinema.films from movieDetails.bson
2024-12-11T11:57:06.304+0100    finished restoring cinema.films (2295 documents,
 0 failures)
2024-12-11T11:57:06.304+0100    2295 document(s) restored successfully. 0 docume
nt(s) failed to restore.
```

zaidouni@zaidouni:~/Documents$ **mongosh**

Pour tester la BD, tapez les commandes :

```
> use cinema;
> db.films.count()
> db.films.findOne()
```

```
test> use cinema;
switched to db cinema
cinema> db.films.count()
DeprecationWarning: Collection.count() is deprecated. Use countDocuments or esti
matedDocumentCount.
2295
cinema> db.films.findOne()
{
  _id: ObjectId('569190ca24de1e0ce2dfcd4f'),
  title: 'Once Upon a Time in the West',
  year: 1968,
  rated: 'PG-13',
  released: ISODate('1968-12-21T05:00:00.000Z'),
  runtime: 175,
  countries: [ 'Italy', 'USA', 'Spain' ],
  genres: [ 'Western' ],
  director: 'Sergio Leone',
```

Gestion des indexes :

1- Affichage des indexes :

```
> db.films.getIndexes()
```

```
cinema> db.films.getIndexes()
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
cinema>
```

2- Création d'indexes :

Voici un exemple de création d'un nouveau index « **genre** » dont le champ doit être obligatoirement présent :

```
>  db.films.createIndex({"genres": 1}, {"sparse": true})
```

```
cinema> db.films.createIndex({"genres": 1}, {"sparse": true})
genres_1
cinema>
```

Affichage des indexes :

```
> db.films.getIndexes()
```

```
cinema> db.films.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { genres: 1 }, name: 'genres_1', sparse: true }
]
cinema>
```

3- Suppression d'indexes :

On peut supprimer un index avec la commande suivante :

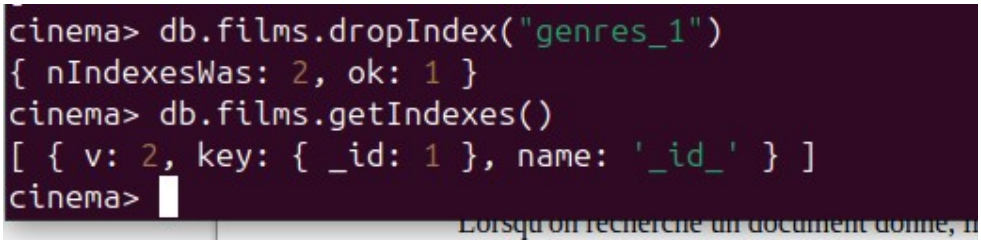
```
> db.collection.dropIndex(name)
```

Pour savoir le nom de l'index à supprimer, utilisez la commande :

```
db.collection.getIndexes()
```

Tapez donc :

```
> db.films.dropIndex("genres_1")
```



Lorsqu'on recherche un document donné, il est possible de connaître la stratégie effective permettant à MongoDB de le retrouver grâce à la commande **`explain()`**.

Dans ce cas MongoDB renvoie un document spécifique qui détaille la recherche avec notamment les indexes utilisés, le nombre de documents parcourus ou encore le temps global de la requête. C'est très pratique quand on veut optimiser ses requêtes ou bien observer l'utilité de ses index.

Exemple :

```
> db.films.find({"actors":"Bruce Willis"}).explain()
```

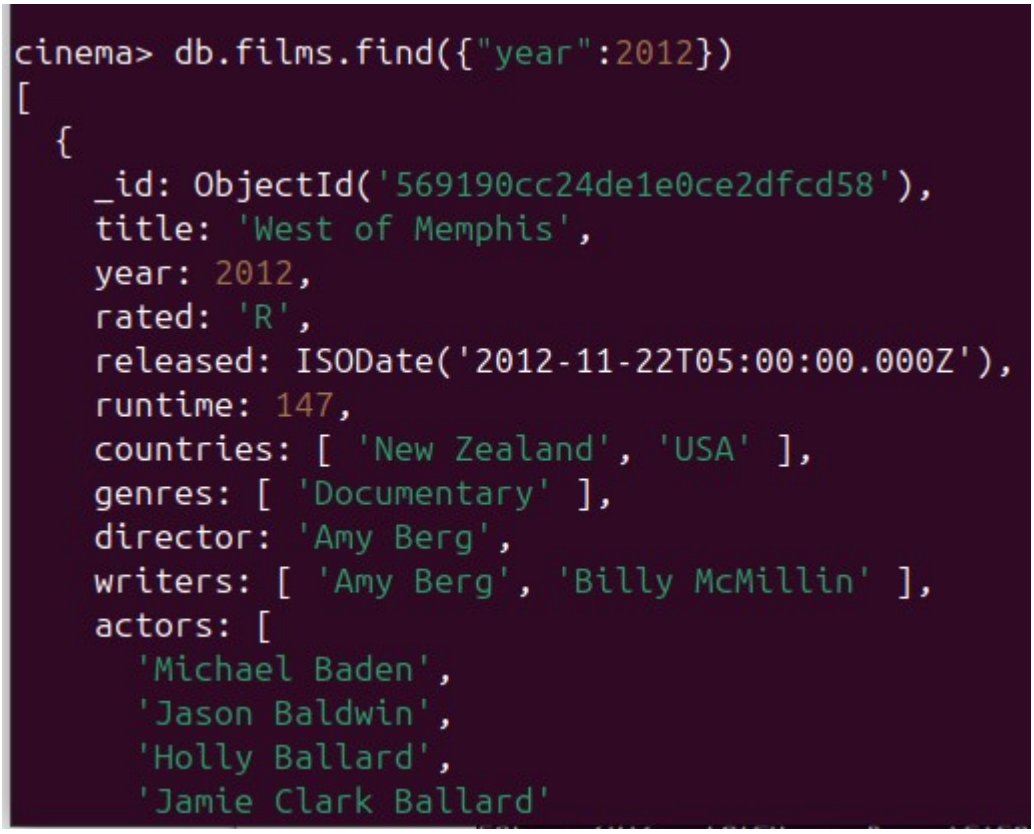
Recherche de documents :

Utilisation d'un filtre simple :

Pour rechercher un document, nous avons besoin de placer une condition dans la commande find. Cette condition est décrite dans un document JSON.

Si on veut par exemple rechercher les films de l'année 2012, il suffit d'écrire la ligne suivante :

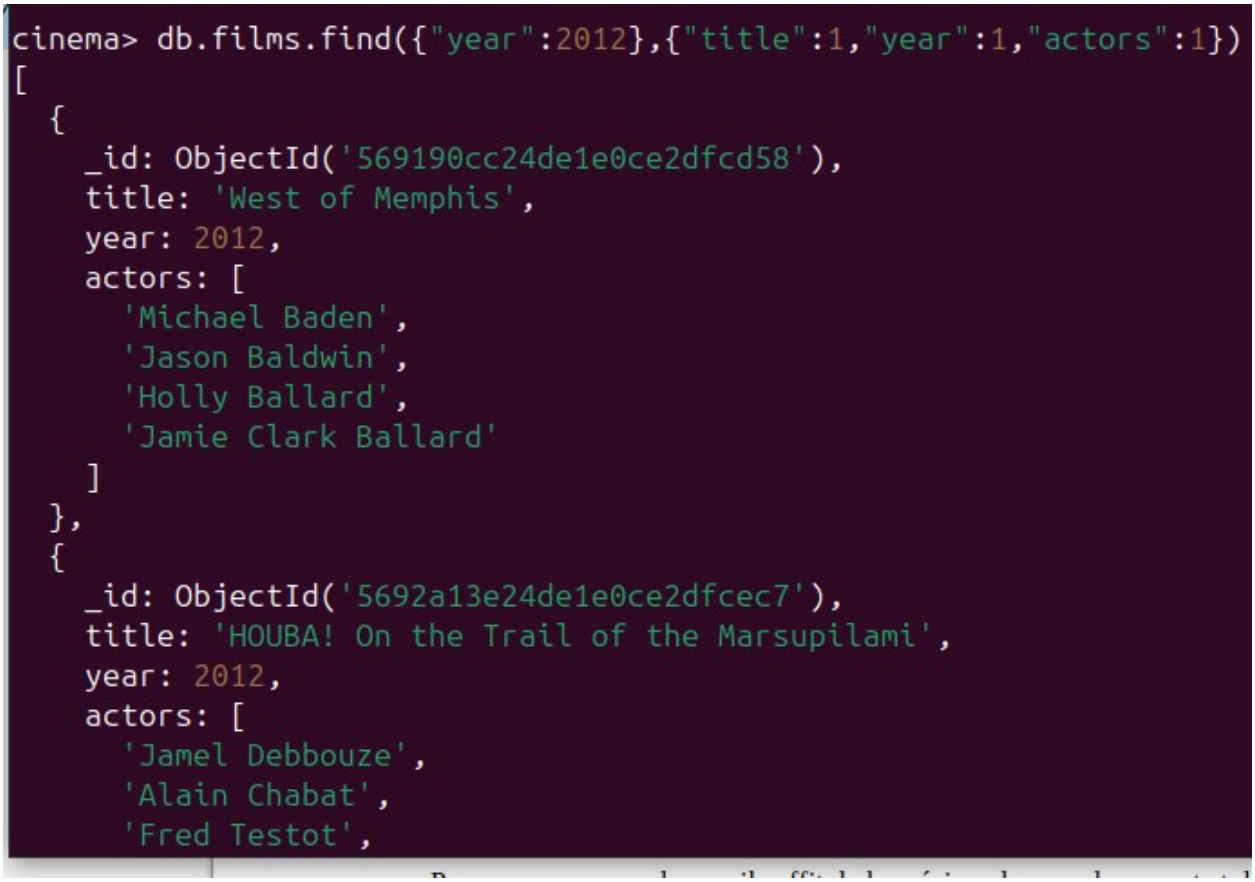
```
> db.films.find({"year":2012})
```



Projection :

La projection permet de sélectionner les informations à renvoyer. Si, par exemple, nous nous intéressons uniquement au titre du film, à son année de sortie et aux noms des acteurs, je vais limiter les informations retournées en précisant les champs souhaités dans un document JSON (toujours ce fameux JSON). Et, également passer ce document comme deuxième argument de ma recherche find.

> db.films.**find({"year":2012},{"title":1,"year":1,"actors":1})**



Pour conserver un champ, il suffit de le préciser dans ce document et de lui affecter la valeur **1**. Vous aurez noté que la requête a renvoyé également le champ `_id`. C'est le fonctionnement normal de la commande find qui renvoie systématiquement la clé du document. Si l'on souhaite l'exclure, il faut le préciser dans la commande.

> **db.films.find({"year":2012},{"title":1,"year":1,"actors":1, _id:0})**

```
cinema> db.films.find({"year":2012},{"title":1,"year":1,"actors":1, _id:0})
[
  {
    title: 'West of Memphis',
    year: 2012,
    actors: [
      'Michael Baden',
      'Jason Baldwin',
      'Holly Ballard',
      'Jamie Clark Ballard'
    ]
  },
  {
    title: 'HOUBA! On the Trail of the Marsupilami',
    year: 2012,
    actors: [
      'Jamel Debbouze',
      'Alain Chabat',
      'Fred Testot',
      'Lambert Wilson'
    ]
```

Recherche dans un tableau :

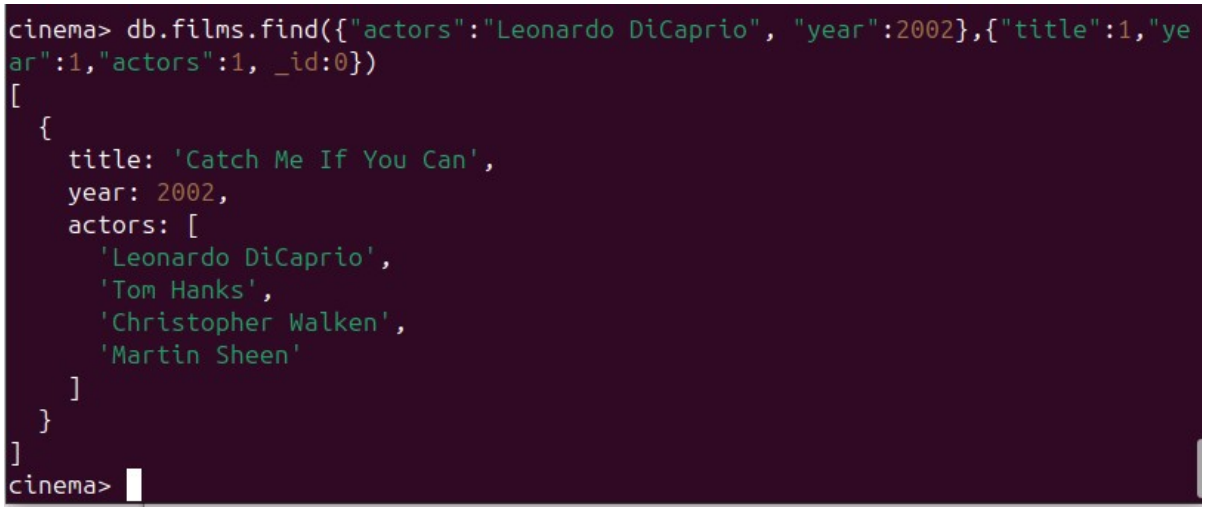
Les documents contiennent des champs simples comme l'année ou le titre du film mais aussi des tableaux pour stockés le nom des acteurs. Comment faire alors des recherches dans un tableau ? Tout simplement de la même manière que pour un champ simple :

```
> db.films.find({"actors":"Leonardo DiCaprio"},
{"title":1,"year":1,"actors":1, _id:0})
```

```
cinema> db.films.find({"actors":"Leonardo DiCaprio"},{"title":1,"year":1,"actors
":1, _id:0})
[
  {
    title: 'Shutter Island',
    year: 2010,
    actors: [
      'Leonardo DiCaprio',
      'Mark Ruffalo',
      'Ben Kingsley',
      'Max von Sydow'
    ]
  },
  {
    title: 'Catch Me If You Can',
    year: 2002,
    actors: [
      'Leonardo DiCaprio',
      'Tom Hanks',
      'Christopher Walken',
      'Martin Sheen'
    ]
```

Vous pouvez combiner plusieurs filtres.

```
> db.films.find({"actors":"Leonardo DiCaprio", "year":2002},
{"title":1,"year":1,"actors":1, _id:0})
```



Nous voulons filtrer les films avec Leonardo DiCaprio **ou** Tom Hanks. Pour cela, nous allons utiliser l'opérateur **$in**.

```
> db.films.find({"actors":{$in:["Leonardo DiCaprio", "Tom \
Hanks"]}},{"title":1,"year":1,"actors":1, _id:0})
```

Pour avoir uniquement les films avec Leonardo DiCaprio **et** Tom Hanks, il existe l'opérateur **$all**.

```
> db.films.find({"actors":{$all:["Leonardo DiCaprio", "Tom \
Hanks"]}},{"title":1,"year":1,"actors":1, _id:0})
```

```
cinema> db.films.find({"actors":{$all:["Leonardo DiCaprio", "Tom Hanks"]}},{"tit
le":1,"year":1,"actors":1, _id:0})
[
  {
    title: 'Catch Me If You Can',
    year: 2002,
    actors: [
      'Leonardo DiCaprio',
      'Tom Hanks',
      'Christopher Walken',
      'Martin Sheen'
    ]
  }
]
cinema>
```

Recherche avancée :

Le langage d'interrogation de MongoDB est extrêmement puissant. Il permet de réaliser toutes les requêtes possibles et l'objectif de ce TP n'est pas d'en faire la liste exhaustive. Nous allons simplement terminer par quelques recherches un peu plus avancées.

Les documents MongoDB peuvent contenir des documents imbriqués. Dans notre collection, nous avons le document `awards`.

```
> db.films.findOne({},{title:1,year:1,awards:1,_id:0})
```

```
test> use cinema;
switched to db cinema
cinema> db.films.count()
DeprecationWarning: Collection.count() is deprecated. Use countDocuments or esti
matedDocumentCount.
2295
cinema> db.films.findOne()
{
  _id: ObjectId('569190ca24de1e0ce2dfcd4f'),
  title: 'Once Upon a Time in the West',
  year: 1968,
  rated: 'PG-13',
  released: ISODate('1968-12-21T05:00:00.000Z'),
  runtime: 175,
  countries: [ 'Italy', 'USA', 'Spain' ],
  genres: [ 'Western' ],
  director: 'Sergio Leone',
```

```
> db.films.find({"awards.wins":7},
{title:1,year:1,awards:1,_id:0})
```

```
cinema> db.films.find({"awards.wins":7}, {title:1,year:1,awards:1,_id:0})
[
  {
    title: 'How the West Was Won',
    year: 1962,
    awards: {
      wins: 7,
      nominations: 5,
      text: 'Won 3 Oscars. Another 7 wins & 5 nominations.'
    }
  },
  {
    title: 'Brazilian Western',
    year: 2013,
    awards: { wins: 7, nominations: 6, text: '7 wins & 6 nominations.' }
  },
  {
    title: 'Where the Wild Things Are',
    year: 2009,
    awards: {
      wins: 7,
```

Dans toutes nos recherches, nous avons utilisé des conditions d'égalité avec l'opérateur **:.** A cause des contraintes JSON, il n'est pas possible d'utiliser les signes habituels (>, >=, <, <=, ! =). MongoDB propose à la place des opérateurs dédiés (**$gt, $gte, $lt, $lte, $ne**).
Prenons un exemple et faisons une recherche sur tous les films ayant remportés au moins un prix.

```
> db.films.find({"awards.wins":{$ne:0}},
{title:1,year:1,awards:1,_id:0})
```

```
cinema> db.films.find({"awards.wins":{$ne:0}},{title:1,year:1,awards:1,_id:0})
[
  {
    title: 'Once Upon a Time in the West',
    year: 1968,
    awards: { wins: 4, nominations: 5, text: '4 wins & 5 nominations.' }
  },
  {
    title: 'Wild Wild West',
    year: 1999,
    awards: { wins: 10, nominations: 11, text: '10 wins & 11 nominations.' }
  },
  {
    title: 'West Side Story',
    year: 1961,
    awards: {
      wins: 18,
      nominations: 11,
      text: 'Won 10 Oscars. Another 18 wins & 11 nominations.'
    }
```

Un autre exemple avec une recherche sur les films ayant remportés plus de 80 prix.

```
> db.films.find({"awards.wins":{$gte:80}},
{title:1,year:1,awards:1,_id:0})
```

Trier les résultats :

Quand les recherches renvoient beaucoup de documents, il est utile de les trier. On dispose pour cela de la fonction **sort** qui va permettre de trier les résultats sur un champ par ordre croissant ou décroissant. L'usage est très simple. Nous allons donc reprendre notre requête précédente et trier les résultats par ordre décroissant sur le nombre de prix obtenus.

```
> db.films.find({"awards.wins":{$gte:80}},
{title:1,year:1,awards:1,_id:0}).sort({"awards.wins":-1})
```

```
cinema> db.films.find({"awards.wins":{$gte:80}},{title:1,year:1,awards:1,_id:0})
.sort({"awards.wins":-1})
[
  {
    title: 'No Country for Old Men',
    year: 2007,
    awards: {
      wins: 148,
      nominations: 122,
      text: 'Won 4 Oscars. Another 148 wins & 122 nominations.'
    }
  },
  {
    title: 'There Will Be Blood',
    year: 2007,
    awards: {
      wins: 103,
      nominations: 123,
      text: 'Won 2 Oscars. Another 103 wins & 123 nominations.'
    }
```

Insértion des documents dans MongoDB :

On va commencer par insérer un document grâce à la fonction insertOne().

```
> use test ;
> db.test.insertOne({"name":"zaidouni"})
```

```
cinema> use test ;
switched to db test
test> db.test.insertOne({"name":"zaidouni"})
{
  acknowledged: true,
  insertedId: ObjectId('675975decc8d3bdc12e9496a')
}
test>
```

Quand on regarde la collection, on peut constater que MongoDB, a bien inséré le document. Il a créé également une clé, nommée *_id.*

> **db.test.find()**

```
test> db.test.find()
[ { _id: ObjectId('675975decc8d3bdc12e9496a'), name: 'zaidouni' } ]
test>
```

Si l'on refait la même insertion :

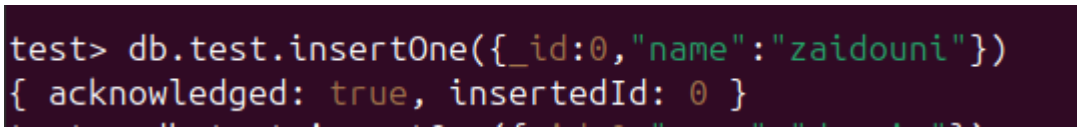
```
> db.test.insertOne({"name":"zaidouni"})
> db.test.find()
```

On obtient le résultat suivant:

MongoDB a inséré un deuxième document. Le système n'a pas détecté de doublon: il a généré une clé différente pour les deux enregistrements.

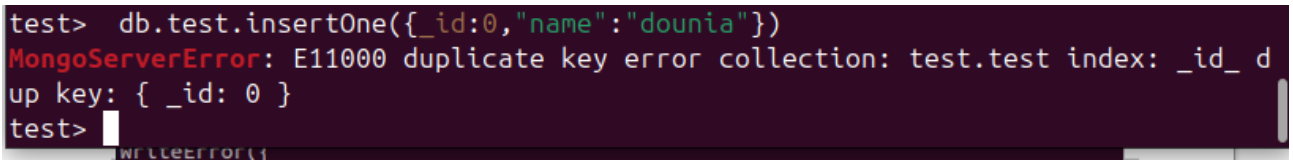
On peut décider de prendre la main et définir nous même la clé ***_id***.

```
> db.test.insertOne({_id:0,"name":"zaidouni"})
```



Donc, si nous essayons cette fois-ci d'insérer un nom avec le même _id:0, cela va générer une erreur.

```
> db.test.insertOne({_id:0,"name":"dounia"})
```



Nous pouvons également insérer un document avec une structure différente dans la même collection.

```
> db.test.insertOne({"name":"toto","age":35})
> db.test.find()
```

```
test> db.test.insertOne({"name":"toto","age":35})
{
  acknowledged: true,
  insertedId: ObjectId('675976d7cc8d3bdc12e9496c')
}
test> db.test.find()
[
  { _id: ObjectId('675975decc8d3bdc12e9496a'), name: 'zaidouni' },
  { _id: ObjectId('67597638cc8d3bdc12e9496b'), name: 'zaidouni' },
  { _id: 0, name: 'zaidouni' },
  { _id: ObjectId('675976d7cc8d3bdc12e9496c'), name: 'toto', age: 35 }
]
test>
```

Il est possible d'insérer plusieurs documents en même temps. On va passer en argument un tableau de documents à la fonction `insert()`.

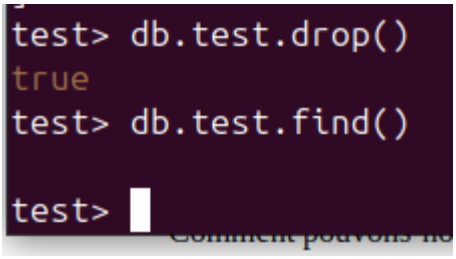
```
> db.test.insertMany([{"name":"mohamed"},{"name": "othman", "ville":["casablanca","rabat"]}])

> db.test.find()
```

```
test> db.test.insertMany([{"name":"mohamed"},{"name": "othman", "ville":["casabl
anca","rabat"]}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('675979c2cc8d3bdc12e9496f'),
    '1': ObjectId('675979c2cc8d3bdc12e94970')
  }
}
test> db.test.find()
[
  { _id: ObjectId('675975decc8d3bdc12e9496a'), name: 'zaidouni' },
  { _id: ObjectId('67597638cc8d3bdc12e9496b'), name: 'zaidouni' },
  { _id: 0, name: 'zaidouni' },
  { _id: ObjectId('675976d7cc8d3bdc12e9496c'), name: 'toto', age: 35 },
  { _id: ObjectId('675979c2cc8d3bdc12e9496f'), name: 'mohamed' },
  {
    _id: ObjectId('675979c2cc8d3bdc12e94970'),
    name: 'othman',
    ville: [ 'casablanca', 'rabat' ]
  }
]
test>
```

Supprimer des documents dans MongoDB :

Pour supprimer une collection, on utilise la fonction drop(). On peut vérifier ensuite que la collection ne contient plus aucun document.

```
> db.test.drop()
> db.test.find()
```

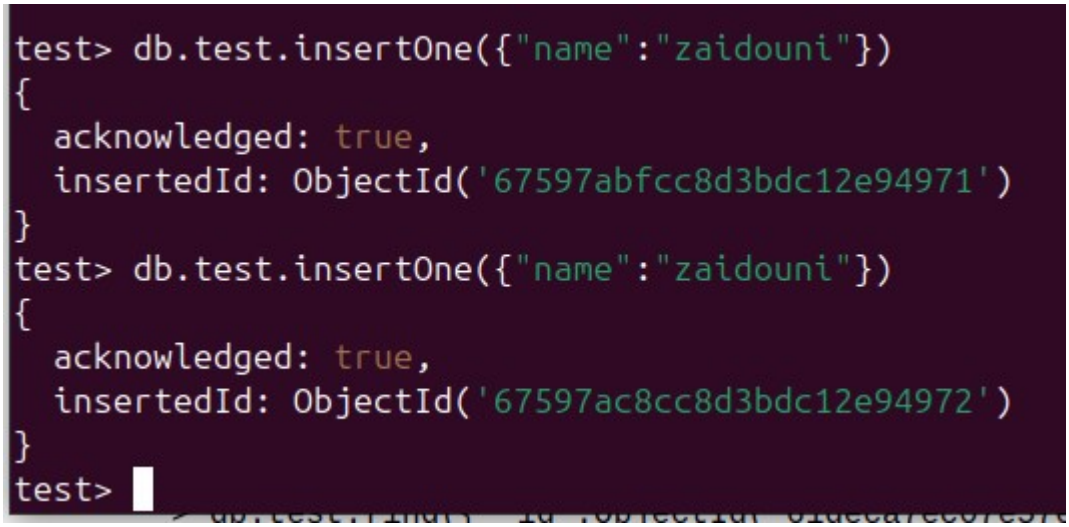


Si on ne souhaite pas supprimer la collection mais uniquement certains documents, on va utiliser la fonction **`deleteOne()`** ou **`deleteMany()`**.

Comment pouvons-nous choisir les documents à supprimer ? Par une condition que l'on va passer comme argument de notre requête.

Nous allons peupler de nouveau notre collection test. On refait les insertions :

```
> db.test.insertOne({"name":"zaidouni"})
> db.test.insertOne({"name":"zaidouni"})
```



Imaginons que nous voulions sélectionner le deuxième document portant le nom « zaidouni ». Nous allons commencer par faire une requête pour sélectionner le document qui nous intéresse. Une fois la condition validée, nous pouvons remplacer la fonction `find()` par la fonction `deleteOne()`.

Tapez donc :

```
> db.test.find({"_id":ObjectId('67597abfcc8d3bdc12e94971')})
[ { _id: ObjectId('67597abfcc8d3bdc12e94971'), name:
'zaidouni' } ]

> db.test.deleteOne({"_id":ObjectId('67597abfcc8d3bdc12e94971')})
```

```
test> db.test.find({"_id":ObjectId('67597abfcc8d3bdc12e94971')})
[ { _id: ObjectId('67597abfcc8d3bdc12e94971'), name: 'zaidouni' } ]
test> db.test.deleteOne({"_id":ObjectId('67597abfcc8d3bdc12e94971')})
{ acknowledged: true, deletedCount: 1 }
test>
```

Mettre à jour des documents dans MongoDB :

MongoDB met à disposition la fonction **update** avec différents opérateurs en fonction du type de mise à jour souhaité. La fonction **update** prend deux arguments obligatoires :

- un document représentant la condition de recherche des documents de la collection,

- un document représentant la mise à jour souhaitée.

1- Ajouter ou remplacer un champ existant avec $set :

> db.test**.updateOne({name:"zaidouni"},{$set:{ville:"rabat"}})**

```
> db.test.find()
```

```
test> db.test.updateOne({name:"zaidouni"},{$set:{ville:"rabat"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
test> db.test.find()
[
  {
    _id: ObjectId('67597ac8cc8d3bdc12e94972'),
    name: 'zaidouni',
    ville: 'rabat'
  }
]
test>
```

Dans cet exemple, on a simplement rajouté un champ ville dans le document de « zaidouni ».

Rajoutons ces documents dans notre collection :

```
> db.test.insertOne({name:"mohamed",age:40})
> db.test.insertOne({name:"zaidouni"})
```

Puis :

```
> db.test.updateOne({name:"zaidouni"},{$set:
{ville:"casablanca"}})
```

```
test> db.test.insertOne({name:"mohamed",age:40})
{
  acknowledged: true,
  insertedId: ObjectId('67597c4bcc8d3bdc12e94973')
}
test> db.test.insertOne({name:"zaidouni"})
{
  acknowledged: true,
  insertedId: ObjectId('67597c53cc8d3bdc12e94974')
}
test> db.test.updateOne({name:"zaidouni"},{$set:{ville:"casablanca"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
> db.test.find()
```

```
test> db.test.find()
[
  {
    _id: ObjectId('67597ac8cc8d3bdc12e94972'),
    name: 'zaidouni',
    ville: 'casablanca'
  },
  {
    _id: ObjectId('67597c4bcc8d3bdc12e94973'),
    name: 'mohamed',
    age: 40
  },
  { _id: ObjectId('67597c53cc8d3bdc12e94974'), name: 'zaidouni' }
]
test>
```

Le résultat de la commande montre qu'un seul document a été mis à jour. La commande find montre que le premier document avec le nom « zaidouni» a bien été modifié mais pas le second. Si vous souhaitez modifier tous les documents, il faut utiliser la fonction `updateMany()`.

```
> db.test.updateMany({name:"zaidouni"},{$set:{ville:"Temara"}})
> db.test.find()
```

```
test> db.test.updateMany({name:"zaidouni"},{$set:{ville:"Temara"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
test> db.test.find()
[
  {
    _id: ObjectId('67597ac8cc8d3bdc12e94972'),
    name: 'zaidouni',
    ville: 'Temara'
  },
  {
    _id: ObjectId('67597c4bcc8d3bdc12e94973'),
    name: 'mohamed',
    age: 40
  },
  {
    _id: ObjectId('67597c53cc8d3bdc12e94974'),
    name: 'zaidouni',
    ville: 'Temara'
  }
]
test>
```

2- Incrémenter un champ numérique existant avec $inc :

Dans certains cas, nous voulons faire une mise à jour en se basant sur la valeur actuelle du champ. **$inc** permet de rajouter une valeur à une donnée numérique. Cette valeur peut être positive ou négative. Si nous souhaitons par exemple incrémenter l'âge de « mohamed », nous pouvons exécuter la commande suivante :

> db.test.updateOne({name:"mohamed"},**{$inc:{age:1}** })

3- Mettre à jour un tableau avec $push ou $pull :

Si nous utilisons $set sur un tableau, nous allons remplacer le tableau existant par un nouveau élément. Comment mettre à jour le tableau sans écraser les données existantes ? L'opérateur **$push** permet de rajouter un nouvel élément à un tableau.

```
> db.test.insertOne({"name": "othman", "ville":
["casablanca","rabat"]})
> db.test.updateOne({"name": "othman"}, {$push:{ville:"tanger"}})
```

```
test> db.test.insertOne({"name": "othman", "ville":["casablanca","rabat"]})
{
  acknowledged: true,
  insertedId: ObjectId('675981b6cc8d3bdc12e94975')
}
test> db.test.updateOne({"name": "othman"}, {$push:{ville:"tanger"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
test> db.test.find()
[
  {
    _id: ObjectId('67597ac8cc8d3bdc12e94972'),
    name: 'zaidouni',
    ville: 'Temara'
  }
```

> db.test.find()

```
test> db.test.find()
[
  {
    _id: ObjectId('67597ac8cc8d3bdc12e94972'),
    name: 'zaidouni',
    ville: 'Temara'
  },
  {
    _id: ObjectId('67597c4bcc8d3bdc12e94973'),
    name: 'mohamed',
    age: 41
  },
  {
    _id: ObjectId('67597c53cc8d3bdc12e94974'),
    name: 'zaidouni',
    ville: 'Temara'
  },
  {
    _id: ObjectId('675981b6cc8d3bdc12e94975'),
    name: 'othman',
    ville: [ 'casablanca', 'rabat', 'tanger' ]
  }
]
test>
```

Dans cet exemple, nous avons rajouté « tanger » dans le tableau contenant déjà « casablanca » et « rabat ». Il faut noter que si « tanger » était déjà présente, l'élément aurait été quand même inséré. Si l'on ne souhaite pas de doublon, il existe l'opérateur **$addToSet** qui assure cette fonction.

Pour supprimer un élément, nous pouvons utiliser **$pull**. Ainsi, si nous souhaitons supprimer la ville de « rabat», il faudra lancer la commande suivante :

```
> db.test.updateOne({"name": "othman"}, {$pull:{ville:"rabat"}})
```

```
test> db.test.updateOne({"name": "othman"}, {$pull:{ville:"rabat"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
> db.test.find()
```

```
test> db.test.find()
[
  {
    _id: ObjectId('67597ac8cc8d3bdc12e94972'),
    name: 'zaidouni',
    ville: 'Temara'
  },
  {
    _id: ObjectId('67597c4bcc8d3bdc12e94973'),
    name: 'mohamed',
    age: 41
  },
  {
    _id: ObjectId('67597c53cc8d3bdc12e94974'),
    name: 'zaidouni',
    ville: 'Temara'
  },
  {
    _id: ObjectId('675981b6cc8d3bdc12e94975'),
    name: 'othman',
    ville: [ 'casablanca', 'tanger' ]
  }
]
test>
```

3) Implémentation d’une application avec Node.js et MongoDB et réalisation des opérations CRUD

Opérations CRUD :

Les opérations CRUD signifie : Create, Read, Update et Delete.
Ceux sont les opérations de bases qu'une application web simple doit réaliser.

Les étapes d’installation de Node.js :

1) Installer NVM (Node Version Manager)

```
$ curl -fsSL \
https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh \
| bash

$ source ~/.bashrc   # ou ferme/rouvre le terminal
```

 # 2) Installer Node 20 (npm inclus)

```
$ nvm install 20
$ nvm use 20
```

 # 3) Vérifier

```
$ node -v
$ npm -v
```

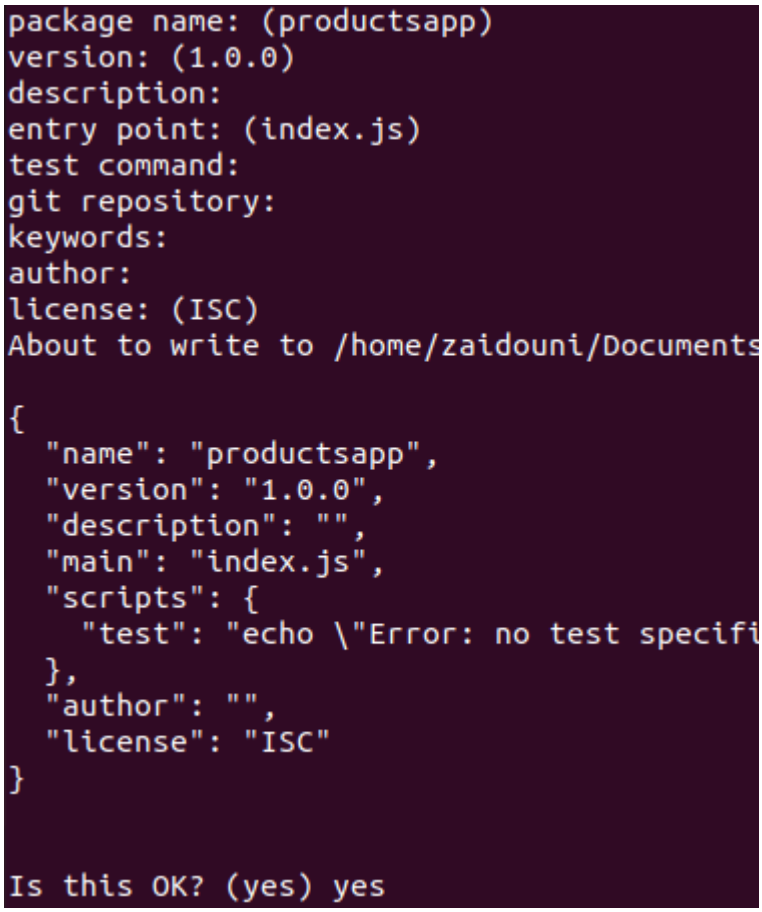
```
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$ nvm install 20
Downloading and installing node v20.19.5...
Downloading https://nodejs.org/dist/v20.19.5/node-v20.19.5-linux-x64.tar.xz...
######################################################################## 100.0%
Computing checksum with sha256sum
Checksums matched!
Now using node v20.19.5 (npm v10.8.2)
Creating default alias: default -> 20 (-> v20.19.5)
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$ nvm use 20
Now using node v20.19.5 (npm v10.8.2)
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$ node -v
v20.19.5
zaidouni@zaidouni-ThinkPad-P14s-Gen-4:~$ npm -v
10.8.2
```

2- Crééz un répertoire appelé « **productsapp** » dans le répertoire : /home/user/Documents (par exemple) :

```
$ cd /home/zaidouni/Documents/
$ mkdir productsapp
```

3- Dans ce répertoire nouvellement créé, exécutez la commande suivante :

```
zaidouni@zaidouni:~/Documents$ cd productsapp/
zaidouni@zaidouni:~/Documents/productsapp$ npm init
```



Les commandes ci-dessus entraînent la création d'un fichier package.json. Le fichier package.json est utilisé pour gérer les packages npm installés localement. Il comprend également les métadonnées sur le projet telles que le nom et le numéro de version.

Installation des packages nécessaires:

Nous allons installer les packages que nous utiliserons pour notre API qui sont :
1- **ExpressJS** : c'est une application Web Node.JS flexible qui a de nombreux fonctionnalités pour les applications Web et mobiles.
2- **mongoose** : c'est la librairie d'Object Data Modeling (ODM) pour MongoDB and Node.js.
3- **body-parser** : package pouvant être utilisé pour gérer les requêtes JSON.

Nous pouvons installer les packages mentionnés ci-dessus en tapant ce qui suit. Assurez-vous simplement que vous êtes dans le répertoire de projet avant d'exécuter la commande ci-dessous.

```
~/Documents/productsapp$
npm install --save express body-parser mongoose
```

Initialisation du serveur :

Créez un nouveau fichier « app.js » directement dans le répertoire de l'application « productsapp » :

```
zaidouni@zaidouni:~/Documents/productsapp$ gedit app.js
```

Ouvrez le nouveau fichier nommé **app.js** et ajoutez toutes les dépendances précédemment installées (ExpressJS et body-parser) en insérant le contenu suivant dans app.js :

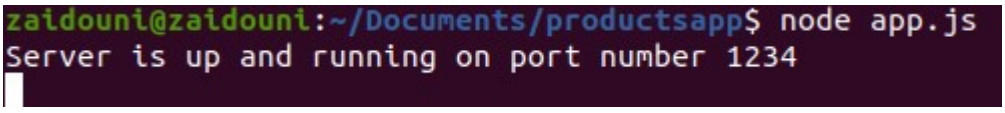
```
// app.js
const express = require('express');
const bodyParser = require('body-parser');

// initialize our express app
const app = express();
```

La prochaine étape serait de dédier un numéro de port et de dire à notre application express d'écouter ce port. Pour cela, ajouter le contenu suivant à la fin du fichier app.js :

```
let port = 1234;
app.listen(port, () => {
console.log('Server is up and running on port number ' + port);
});
```

Maintenant, nous devrions pouvoir tester notre serveur en utilisant la commande suivante dans le terminal :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```



Maintenant, nous avons un serveur opérationnel. Cependant, ce serveur ne fait rien!
Nous allons par la suite rendre notre application plus complexe.

Organisation de l'application :

Nous travaillerons avec un modèle de conception appelé MVC. C'est une bonne façon de séparer les parties de notre application et de les regrouper en fonction de leur fonctionnalité et de leur rôle.
M signifie modèles (Models), cela inclura tous les code pour nos modèles de base de données

(qui dans ce cas seront des produits). Ensuite le **V** qui représente les vues (views) et layout. Nous ne couvrirons pas les vues dans ce TP puisque nous concevons une API. La partie restante est le **C**, qui signifie contrôleurs (controllers), qui est la logique de comment l'application gère les demandes
entrantes et les réponses sortantes.
Il y a aussi autre chose, appelée **Routes**, elles indiquent au client (navigateur / application mobile) d'aller vers quel contrôleur une fois qu'une URL / un chemin spécifique est demandé.
Dans le répertoire productsapp, nous allons créer les quatre sous-répertoires suivants :
1- controllers
2- models
3- routes
4- views

Pour cela, tapez les commandes suivantes :

```
zaidouni@zaidouni:~/Documents/productsapp$ mkdir controllers
zaidouni@zaidouni:~/Documents/productsapp$ mkdir models
zaidouni@zaidouni:~/Documents/productsapp$ mkdir routes
zaidouni@zaidouni:~/Documents/productsapp$ mkdir views
```

Nous avons maintenant un serveur qui est prêt à gérer nos demandes et certains répertoires qui contiendraient le code.

Models :

Nous allons commencer par définir notre modèle. Créez un nouveau fichier dans le répertoire « **models** » et appelons-le « **product.model.js** » :

```
~/Documents/productsapp$ cd models/
~/Documents/productsapp/models$ gedit product.model.js
```

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

let ProductSchema = new Schema({
name: {type: String, required: true, max: 100},
price: {type: Number, required: true},
});

// Export the model
module.exports = mongoose.model('Product', ProductSchema);
```

Dans ce code, nous avons commencé par les exigences de mongoose, puis nous avons défini le schéma de notre modèle. La dernière chose est d'exporter le modèle pour qu'il puisse être utilisé par d'autres fichiers de notre projet.
Maintenant, nous avons terminé avec la partie Models.

Routes :

Dans le répertoire routes, créez un fichier « **product.route.js** ». Il s'agit du fichier qui contiendra les routes des produits.

```
~/Documents/productsapp/models$ cd ..
~/Documents/productsapp$ cd routes/
~/Documents/productsapp/routes$ gedit product.route.js
```

Copiez le contenu suivant dans **product.route.js :**

```
const express = require('express');
const router = express.Router();

// Require the controllers WHICH WE DID NOT CREATE YET!!
const product_controller = require('../controllers/product.controller');

// a simple test url to check that all of our files are communicating correctly.
router.get('/test', product_controller.test);
module.exports = router;
```

Controllers :

L'étape suivante consiste à implémenter les contrôleurs que nous avons référencés dans les routes. Nous allons donc créer un nouveau fichier js nommé **product.controller.js** qui sera l'espace réservé pour nos contrôleurs qui sera placé dans le répertoire controllers.

```
~/Documents/productsapp/routes$ cd ..
~/Documents/productsapp$ cd controllers/
~/Documents/productsapp/controllers$ gedit product.controller.js
```

```
const Product = require('../models/product.model');

//Simple version, without validation or sanitation
exports.test = function (req, res) {
res.send('Greetings from the Test controller!');
};
```

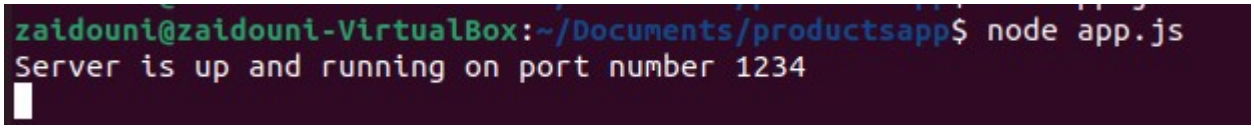
La dernière étape avant d'essayer notre première route consiste à ajouter les lignes en gras relatif à l'ajout de la classe routes à **app.js** :

```
~/Documents/productsapp/controllers$ cd ..
~/Documents/productsapp$ gedit app.js
```

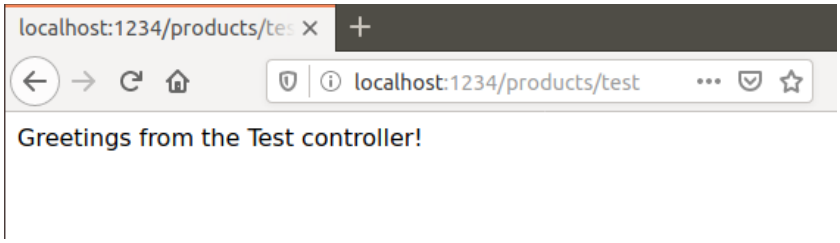
```
// app.js
const express = require('express');
const bodyParser = require('body-parser');

const product = require('./routes/product.route'); // Imports routes for the products
const app = express();
app.use('/products', product);

const port = 1234;
app.listen(port, () => {
    console.log('Server is up and running on port number ' + port);
});
```

Tapez la commande suivante :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```



Dirigez-vous maintenant vers votre navigateur et essayez le lien suivant : http://localhost:1234/products/test



Postman :

Postman est un client HTTP très puissant utilisé pour les tests, la documentation et le développement d'API. Nous utiliserons Postman ici pour tester nos endpoints que nous allons implémenter dans ce TP. Mais tout d'abord, nous allons installer et tester Postman.

1- Installation de Postman :

Ouvrez un nouveau terminal et tapez :

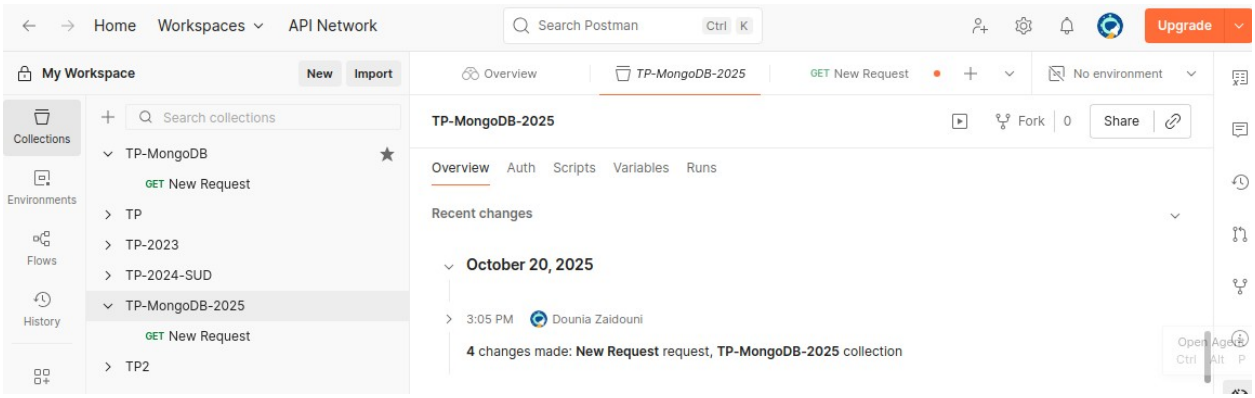
```
zaidouni@zaidouni:~/Documents/productsapp$
$ sudo snap install postman
```

```
zaidouni@zaidouni-VirtualBox:~$ sudo snap install postman
[sudo] Mot de passe de zaidouni :
postman (v11/stable) 11.21.0 from Postman, Inc. (postman-inc✓) installed
zaidouni@zaidouni-VirtualBox:~$
```

Après l'installation, taper « postman » pour pouvoir l'utiliser : **$ postman**

Vous pouvez faire le « **SignIn** » avec votre compte google « Sign In with google ».

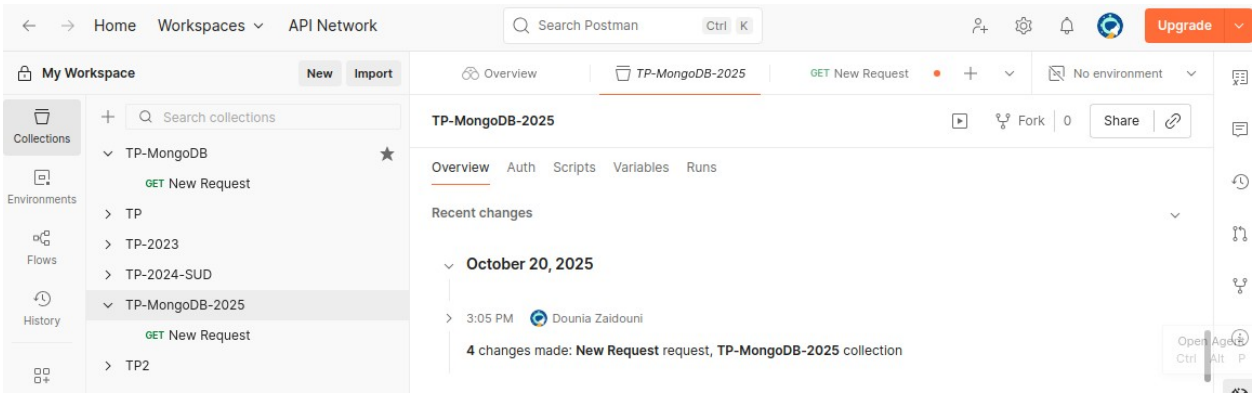
Cliquez sur « Workspaces » puis « Myworkspace », ensuite créez une nouvelle collection avec « New collection » puis « Blank collection » et nommez cette collection (par exemple : TP-MongoDB-2025) et après cliquer sur « Add Request » qui est en bleu.



Ensuite, choisissez « **GET** » comme request et coller l'url suivant :

« **localhost:1234/products/test** » puis cliquez sur « SEND ». Assurez-vous que votre serveur fonctionne toujours sur le numéro de port 1234.

Vous devriez pouvoir voir 'Greetings from Test controller ".



Connection de l’application à la base de données MongoDB :

Connection mongoose:

Nous devons informer notre application qu'elle doit communiquer avec la base de données que nous avons précédemment créée dans la première partie de ce TP. Pour cela nous allons utiliser le package déjà installé de « mongoose ».

Il suffit de nous diriger vers le fichier **app.js** et d'y coller le code suivant **avant** la ligne :
app.use('/products', product);

```
….
// Set up mongoose connection
var mongoose = require('mongoose');
var dev_db_url = 'mongodb://127.0.0.1/tp_database';
var mongoDB = process.env.MONGODB_URI || dev_db_url;
mongoose.connect(mongoDB, { useNewUrlParser: true });
mongoose.Promise = global.Promise;
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));
```

Body Parser :

La dernière chose dont nous avons besoin pour notre configuration est d'utiliser le « bodyParser ».
Body Parser est un package npm utilisé pour analyser les request bodies dans un middleware.

Dans le fichier app.js, ajoutez les deux lignes suivantes juste avant la ligne :
app.use('/products', product);

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));
```

Voici à quoi ressemble notre fichier app.js complet :

```
// app.js
const express = require('express');
const bodyParser = require('body-parser');

const product = require('./routes/product.route'); // Imports routes for the products
const app = express();

// Set up mongoose connection
var mongoose = require('mongoose');
var dev_db_url = 'mongodb://127.0.0.1/tp_database';
var mongoDB = process.env.MONGODB_URI || dev_db_url;
mongoose.connect(mongoDB, { useNewUrlParser: true });
mongoose.Promise = global.Promise;
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

app.use('/products', product);

const port = 1234;
app.listen(port, () => {
console.log('Server is up and running on port number ' + port);
});
```

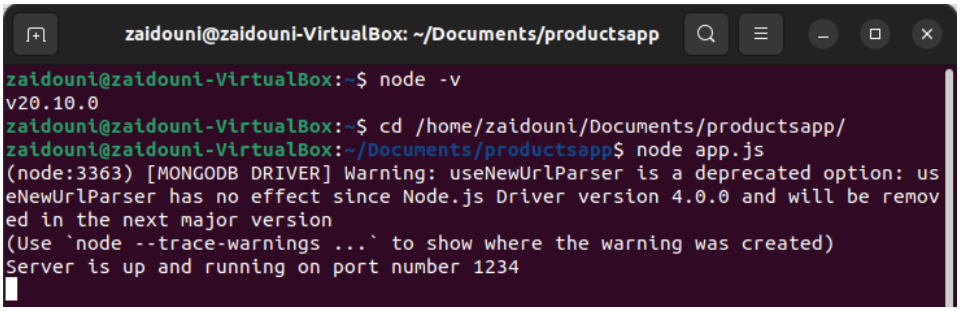
Tapez la commande suivante :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```

Remarque : Si vous obtenez une erreur, vérifiez la version de nodeJS qui est installé avec la commande : `$ node -v`
Si la version de node est une version ancienne, il faut la désinstaller afin de restaurer la version récente nécessaire qui est : **v20.10.0**, pour cela tapez :

```
sudo apt remove nodejs
sudo apt-get autoremove
```

Vérifiez avec node -v et relancez : `$ node app.js`



Implémentation des Endpoints :

CREATE :

La première tâche de nos opérations CRUD est de créer un nouveau produit. Commençons par définir d'abord notre route. Dirigez-vous vers products.route.js et commencez à concevoir le chemin attendu que le navigateur atteindrait et le contrôleur qui serait responsable de la gestion de cette demande.

Ajouter cette ligne dans **products.route.js,** juste **avant** la ligne : **module.exports = router;**

```
// routes/products.route.js
...
router.post('/create', product_controller.product_create);
```

Écrivons maintenant le contrôleur product_create dans notre fichier « product.controller.js ». Ajoutez les lignes suivantes à **la fin** du fichier « **product.controller.js** ».

```
// controllers/products.js
…

exports.product_create = function (req, res, next) {
	let product = new Product(
		{
		  name: req.body.name,
		  price: req.body.price
		}
	);

	product.save().then(result =>  {
	    res.send('Product Created successfully')
	 })
	.catch(function (err) {
	     console.log(err);
	});
}
```

La fonction consiste simplement à **créer** un nouveau produit à l'aide des données provenant d'une demande POST et à l'enregistrer dans notre base de données.
La dernière étape serait de valider que nous pouvons facilement créer un nouveau produit.

Tapez la commande pour prendre en compte les modifications :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```

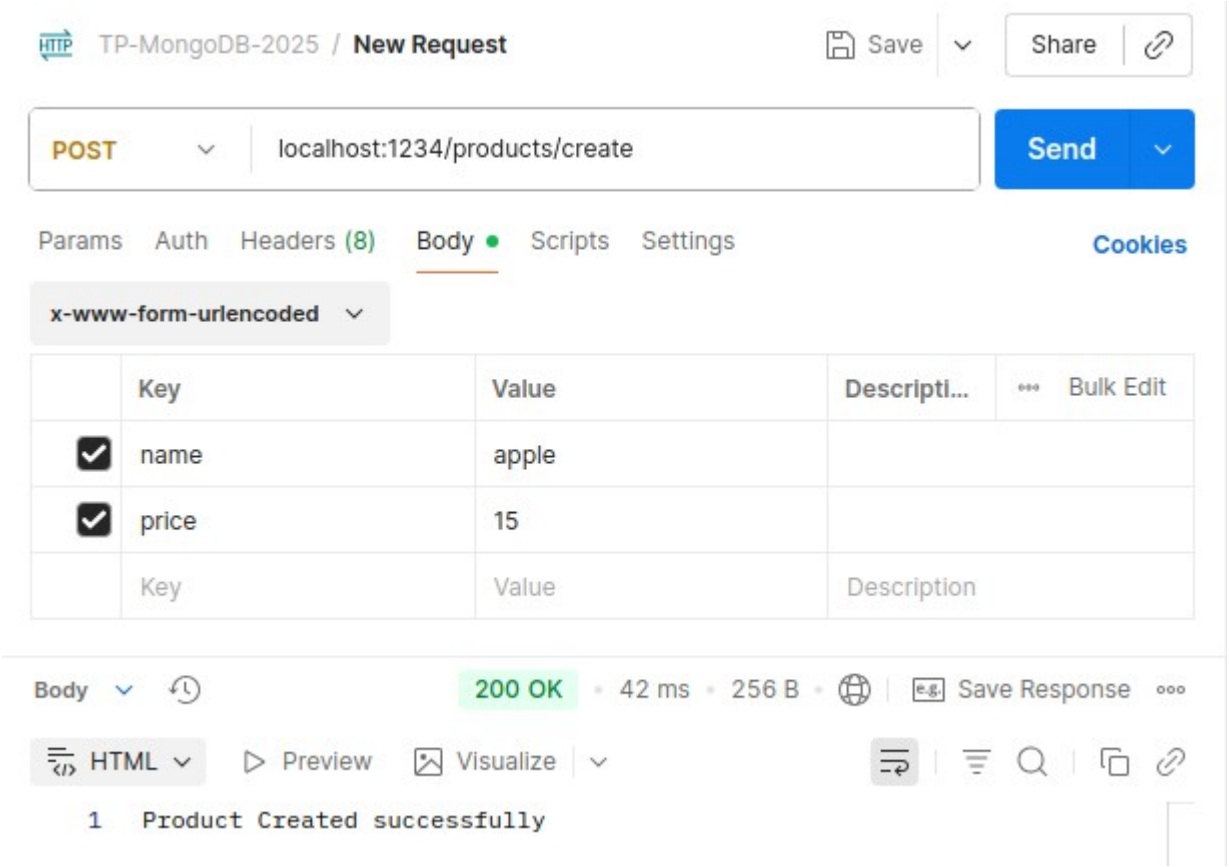
Ouvrons Postman, en tapant la commande : `$postman`

Envoyons ensuite une demande « **POST** » à l'URL suivante:« **localhost:1234/products/create** » et choisissez l'onglet « **Body** » et assurez-vous également que vous choisissez : **x-www-form-urlencoded**

Spécifiez les données POST, par exemple :

name: apple

price: 15

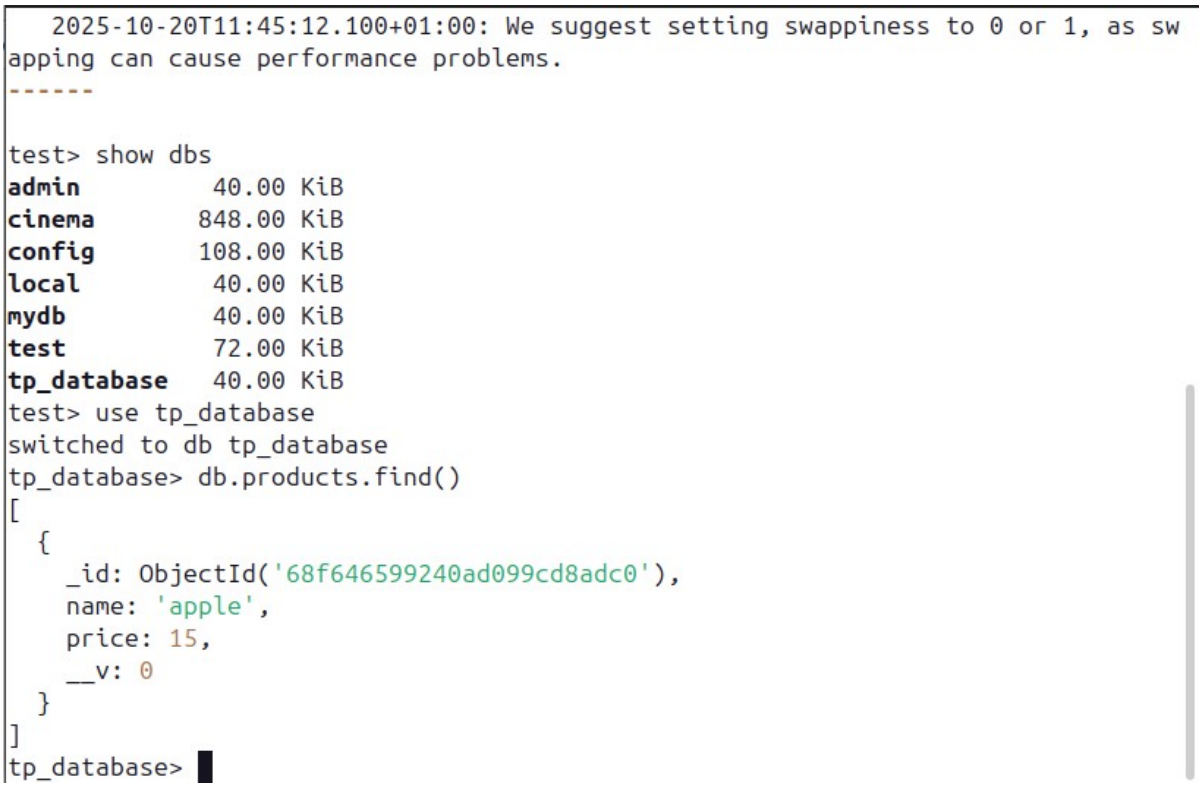


Nous pouvons voir que la réponse est «**Product Created successfully** ». Cela signifie que le routeur et le contrôleur fonctionnent correctement. Pour vérifier à nouveau qu'un produit «apple» a été créé, vérifions la base de données.

Pour cela, connectez-vous au terminal de Mongo et vérifier qu'une table nommé : « tp_database » et une collection « products » sont bien créés .
Dans un nouveau terminal, tapez : `$ mongosh`

Puis :

```
>show dbs
> use tp_database
> db.products.find()
```



Read :

La deuxième tâche de notre application CRUD consiste à **lire** un produit existant.
Pour configurer la route :

Ajouter cette ligne dans products.route.js, juste avant la ligne : module.exports = router; :

```
// routes/products.route.js
...
router.get('/:id', product_controller.product_details);
```

Écrivons maintenant le contrôleur product_details dans notre fichier contrôleur. Ajoutez les lignes suivantes à la fin du fichier « product.controller.js » :

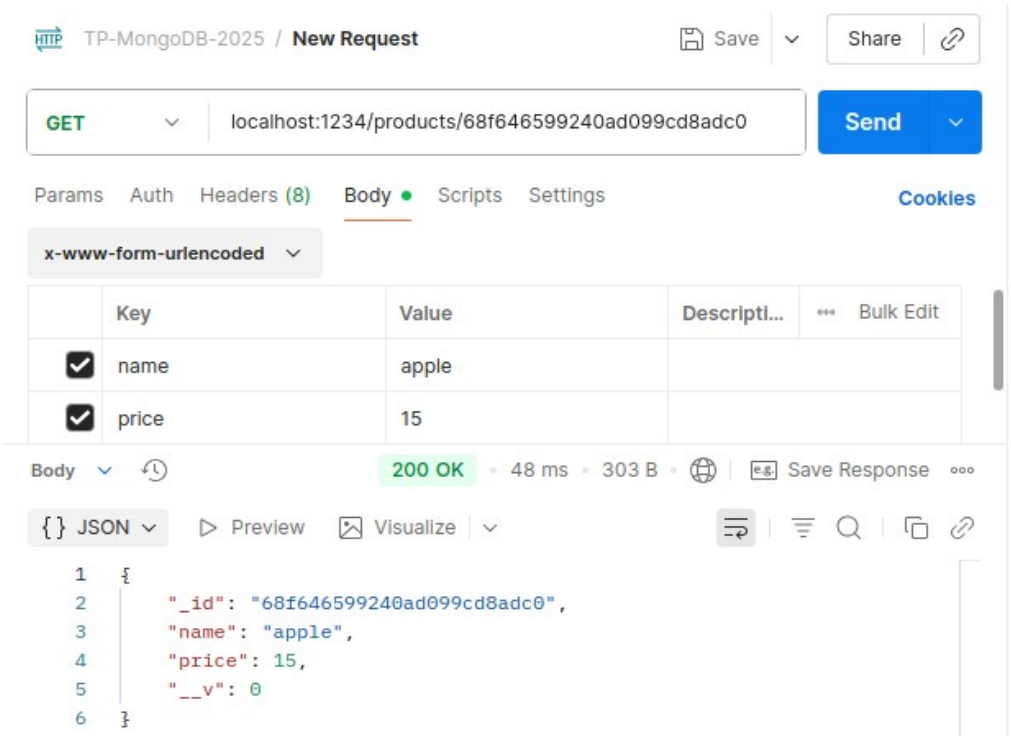
```
// controllers/products.controller.js
…

exports.product_details = function (req, res, next) {
        Product.findById(req.params.id).then(product => {
            res.send(product);
        })
        .catch(function (err) {
            console.log(err);
        });

};
```

La fonction consiste simplement à lire un produit existant à partir de l'ID de produit envoyé dans la demande.
Maintenant, il faut relancer la commande pour prendre en compte les modifications :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```

Ouvrez Postman et essayez notre nouveau Endpoint. Choisissez « **GET** » et appelez l'URL suivant : « **localhost:1234/products/PRODUCT_ID** »

PRODUCT_ID est l'ID de l'objet que nous avons créé dans le Endpoint précédent. Vous devriez l'obtenir de votre base de données et ce sera certainement différent du mien qui est : « **68f646599240ad099cd8adc0** ».



Nous avons obtenu une réponse contenant toutes les informations de ce produit spécifique. Vous pouvez voir qu'il s'appelle « apple » et que son « price » est 15.

Update :

La troisième tâche de notre application CRUD est de mettre à jour un produit existant.
Pour configurer la route :
Ajouter cette ligne dans products.route.js, juste avant la ligne : module.exports = router; :

```
// routes/products.route.js
...
router.put('/:id/update',
product_controller.product_update);
```

Écrivons maintenant le contrôleur product_update dans notre fichier contrôleur.
Ajoutez les lignes suivantes à la fin du fichier « product.controller.js » :

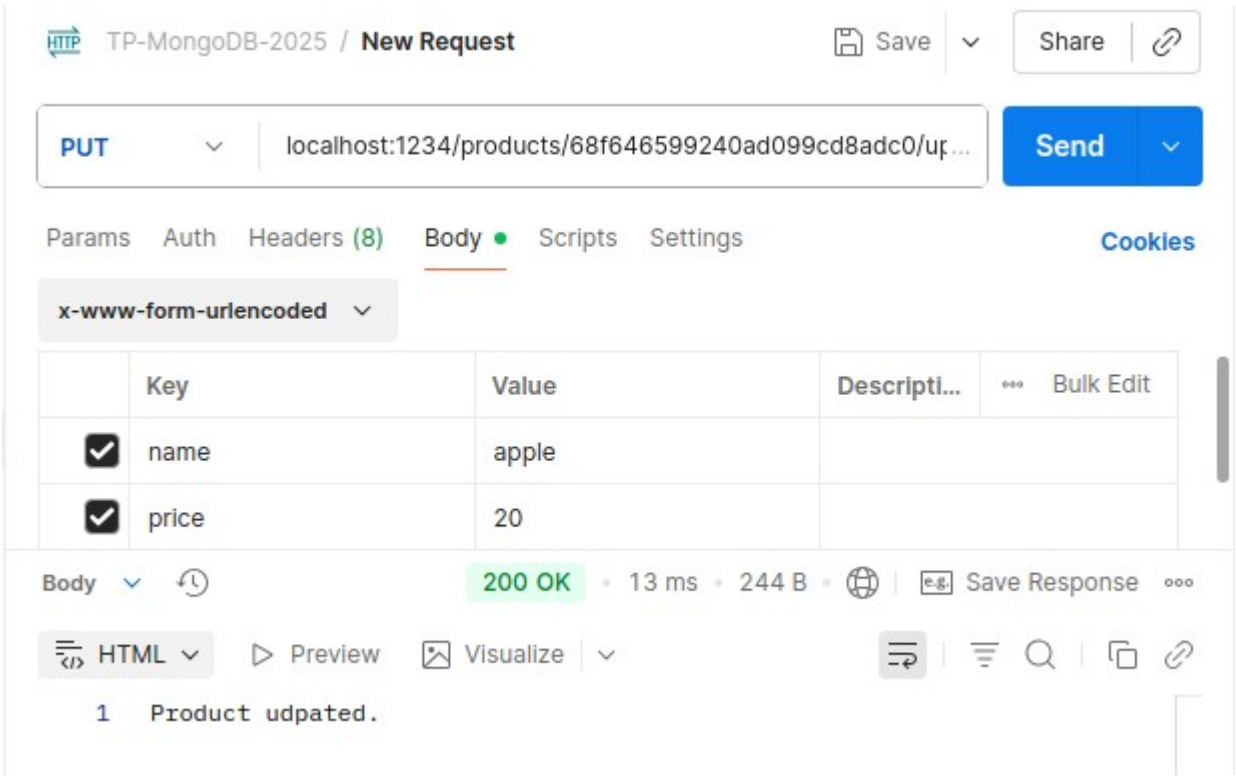
```
// controllers/products.controller.js
…

exports.product_update = function (req, res,next) {
      Product.findByIdAndUpdate(req.params.id, {$set:req.body}).then(product => {
         res.send('Product udpated.');
      })
      .catch(function (err) {
           console.log(err);
      });
};
```

La fonction détecte simplement un produit existant en utilisant son identifiant envoyé dans la demande.
Maintenant, il faut relancer la commande pour prendre en compte les modifications :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```

Ouvrez Postman et essayez notre nouveau Endpoint. Choisissez « **PUT** » et appelez l'URL suivant :
« **localhost:1234/products/PRODUCT_ID/update** »

PRODUCT_ID est l'ID de l'objet que nous avons créé dans le point de terminaison précédent. Vous devriez l'obtenir de votre base de données et ce sera certainement différent du mien.



Pour tester ces modification, connectez-vous au terminal mongosh et vérifier que la collection products a été bien modifiée.

Pour cela taper :

```
$mongosh
```

Puis :

```
> use tp_database
> db.products.find()
```

```
tp_database> db.products.find()
[
  {
    _id: ObjectId('68f646599240ad099cd8adc0'),
    name: 'apple',
    price: 20,
    __v: 0
  }
]
tp_database>
```

Delete :

La dernière tâche de notre application CRUD est de supprimer un produit existant.
Pour configurer la route :
Ajouter cette ligne dans products.route.js, juste avant la ligne : module.exports = router; :

```
// routes/products.route.js
...
router.delete('/:id/delete',
product_controller.product_delete);
```

Écrivons maintenant le contrôleur product_update dans notre fichier contrôleur. Ajoutez les lignes suivantes à la fin du fichier « product.controller.js » :

```
// controllers/products.controller.js
…

exports.product_delete = function (req, res, next) {
        Product.deleteOne({_id:req.params.id}).then(product => {
            res.send('Deleted successfully!');
        })
        .catch(function (err) {
             console.log(err);
        });
};
```

La fonction supprime simplement un produit existant.

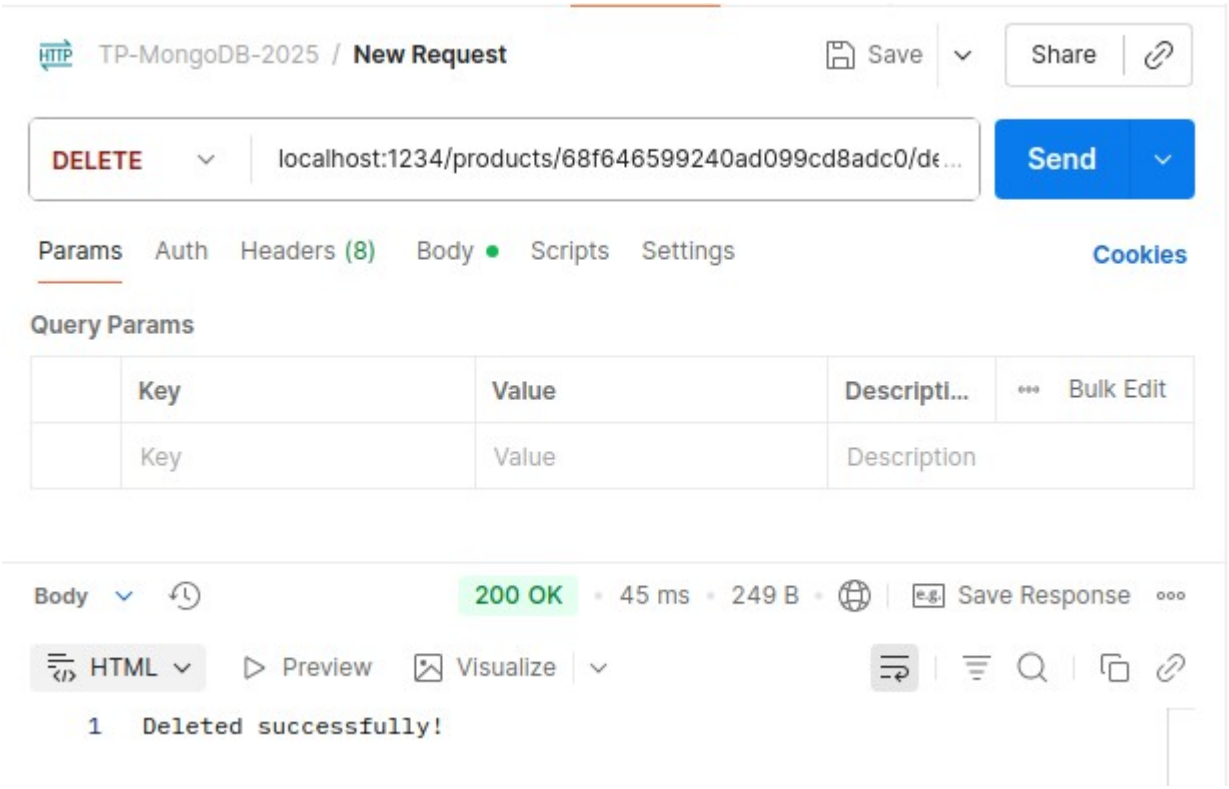
Maintenant, il faut relancer la commande pour prendre en compte les modifications :

```
zaidouni@zaidouni:~/Documents/productsapp$ node app.js
```

Ouvrez Postman et essayez notre nouveau Endpoint. Choisissez « **DELETE** » et appelez l'URL suivant :

«**localhost:1234/products/PRODUCT_ID/delete**»

PRODUCT_ID est l'ID de l'objet que nous avons créé dans le point de terminaison précédent. Vous devriez l'obtenir de votre base de données et ce sera certainement différent du mien.



Pour tester ces modification, connectez-vous au terminal mongosh et vérifiez que la collection products a été bien supprimée.

Pour cela taper : `$ mongosh`
Puis :

```
> use tp_database
> db.products.find()
```

Rapport (à faire par binôme) :

Rédigez un rapport détaillé contenant les captures d'écran et les explications des différentes manipulations effectuées dans ce TP1 .

Bon courage