TP 2 : Installation et manipulation de MongoDB

Filière : Data Engineer, INE2

Encadré par : Pr. D. ZAIDOUNI

Réalisé par : **ISBAINE** Mohammed, **LABRIJI** Saad

Introduction générale au fonctionnement de MongoDB :

MongoDB est un programme de gestion de base de données **NoSQL** open source. **NoSQL** est utilisé comme alternative aux bases de données relationnelles traditionnelles. Les bases de données **NoSQL** sont très utiles pour travailler avec de grands ensembles de données distribuées. **MongoDB** est un outil qui peut gérer des informations orientées document, stocker ou récupérer des informations.

MongoDB prend en charge diverses formes de données. C'est l'une des nombreuses technologies de bases de données non relationnelles apparues au milieu des années 2000 sous la bannière **NoSQL** - normalement, pour une utilisation dans des applications de **Big Data** et d'autres tâches de traitement impliquant des données qui ne s'intègrent pas bien dans un modèle relationnel rigide. Au lieu d'utiliser des tables et des lignes comme dans les bases de données relationnelles, l'architecture **MongoDB** est composée de **collections** et de **documents**.

MongoDB utilise des enregistrements constitués de documents contenant une structure de données composée de paires de champs et de valeurs. Les documents sont l'unité de base des données dans **MongoDB**. Les documents sont similaires à **JavaScript Object Notation**, mais utilisent une variante appelée **Binary JSON** (**BSON**). L'avantage d'utiliser **BSON** est qu'il accepte plus de types de données. Les champs de ces documents sont similaires aux colonnes d'une base de données relationnelle. Les valeurs contenues peuvent être une variété de types de données, y compris d'autres documents, des tableaux et des tableaux de documents, selon le manuel d'utilisation de **MongoDB**. Les documents incorporeront également une clé primaire comme identifiant unique.

Les ensembles de documents sont appelés **collections**, qui fonctionnent comme l'équivalent des tables de bases de données relationnelles. Les collections peuvent contenir n'importe quel type de données, mais la restriction est que les données d'une collection ne peuvent pas être réparties sur différentes bases de données.

Le **Shell mongo** est un composant standard des distributions open source de **MongoDB**. Une fois **MongoDB** installé, les utilisateurs connectent le **Shell mongo** à leurs instances **MongoDB** en cours d'exécution. Le **Shell mongo** agit comme une interface **JavaScript** interactive pour **MongoDB**, qui permet aux utilisateurs d'interroger et de mettre à jour des données et d'effectuer des opérations administratives.

Une représentation binaire des documents de type **JSON** est fournie par le format de stockage de documents et d'échange de données **BSON**. Le partitionnement automatique est une autre fonctionnalité clé qui permet aux données d'une collection **MongoDB** d'être distribuées sur plusieurs systèmes pour une évolutivité horizontale, à mesure que les volumes de données et les exigences de débit augmentent.

Le **SGBD NoSQL** utilise une architecture maître unique pour la cohérence des données, avec des bases de données secondaires qui conservent des copies de la base de données primaire. Les opérations sont automatiquement répliquées sur ces bases de données secondaires pour un basculement automatique.

Les objectifs de ce TP sont les suivants :

- Installation et configuration de MongoDB.
- Examination des fonctionnalités de requêtage de MongoDB.
  - Restauration d'un fichier «.bson » ,
  - Gestion des indexes,
  - Recherche et tri des documents,
  - Insertion, suppression et mise à jour des documents.
- Implémentation d'une application avec Node.js et MongoDB et réalisation des opérations CRUD (create, read, update et delete).

1) Installation et configuration de MongoDB :

Installation et configuration :

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, on suit les étapes suivantes :

Étape 1 - Configuration du « apt Repository »

Tout d'abord, on importe la clé GPK pour le « **MongoDB apt Repository** » sur notre système à l'aide de la commande suivante :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ wget -qO - https://www.mongodb.org/static/pgp/server-5.
0.asc | sudo apt-key add -
OK
```

Ensuite, on crée le fichier liste **/etc/apt/sources.list.d/mongodb-org-5.0.list** pour notre version d'Ubuntu, on tape :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org
/apt/ubuntu focal/mongodb-org/5.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list
deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0 multiverse
```

Étape 2 – Installation de MongoDB sur une VM Ubuntu :

Après avoir ajouté les « **apt Repository** » requis, on utilise les commandes suivantes pour installer **MongoDB** sur notre **VM**. Il installera également tous les packages dépendants requis pour **MongoDB**.

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo apt update
Atteint :1 http://ma.archive.ubuntu.com/ubuntu focal InRelease
Réception de :2 http://ma.archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Réception de :3 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Ign :4 https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0 InRelease
```

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo apt-get install -y mongodb-org
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets supplémentaires suivants seront installés :
  mongodb-database-tools mongodb-mongosh mongodb-org-database mongodb-org-database-tools-extra
```

Affichage de la version de MongoDB :

Pour Afficher, la version de **MongoDB** que nous avons installé, on tape :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ mongod --version
db version v5.0.5
Build Info: {
    "version": "5.0.5",
    "gitVersion": "d65fd89df3fc039b5c55933c0f71d647a54510ae",
    "openSSLVersion": "OpenSSL 1.1.1f  31 Mar 2020",
    "modules": [],
    "allocator": "tcmalloc",
    "environment": {
        "distmod": "ubuntu2004",
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
```

La version de **MongoDB** que nous avons installé est **V5.0.5**

Gestion des services Mongod :

1- Activation et démarrage des services MongoDB :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo systemctl enable mongod
Created symlink /etc/systemd/system/multi-user.target.wants/mongod.service → /lib/systemd/system/mongod.service.
```

On peut démarrer le processus mongod en exécutant la commande suivante :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo systemctl start mongod
```

2- Affichage du statut de mongod :

Vérification que MongoDB a démarré avec succès :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo systemctl status mongod
● mongod.service - MongoDB Database Server
     Loaded: loaded (/lib/systemd/system/mongod.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2022-01-20 12:26:57 +01; 41s ago
       Docs: https://docs.mongodb.org/manual
   Main PID: 4452 (mongod)
     Memory: 62.3M
     CGroup: /system.slice/mongod.service
             └─4452 /usr/bin/mongod --config /etc/mongod.conf

12:26:57 20 يناير labriji-VirtualBox systemd[1]: Started MongoDB Database Server.
```

3- Arrêt et redémarrage des services MongoDB :

On peut utiliser les commandes suivantes pour arrêter ou redémarrer le service **MongoDB** :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo systemctl stop mongod
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo systemctl restart mongod
```

4- Test de la configuration :

On se connecte à **MongoDB** à l'aide de la ligne de commande **$ mongo** , puis on exécute les commandes de test pourvérifier le bon fonctionnement. On exécute la commande suivante :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ mongo
MongoDB shell version v5.0.5
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("ce99d8f5-95e3-4601-bac1-b44995a3adac") }
MongoDB server version: 5.0.5
================
```

Puis on exécute les commandes suivantes :

```
> use mydb;
switched to db mydb
> db.test.save( { isbaine_labriji: 100 } )
WriteResult({ "nInserted" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e947f0d89705615a1c0374"), "isbaine_labriji" : 100 }
>
```

Remarque : La différence avec **.save()** est que si le document passé contient un champ **_id**, si un document existe déjà avec ce champ **_id** il sera mis à jour au lieu d'être ajouté en tant que nouveau.

Examination des fonctionnalités de requêtage de MongoDB :

Restauration d'un fichier « .bson » :

Dans ce TP, nous utilisons les données stockées dans le fichier « **moviedetails.bson** » relatives aux informations sur plusieurs films. On récupère ce fichier à partir de **Moodle**, et déposez-le dans le répertoire **/home/labriji/Bureau** :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ ls
movieDetails.bson  test.txt  TP2_MongoDB.pdf
```

Ensuite, on exécute la commande suivante pour restaurer ces données dans la BD **MongoDB** « cinema » :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo mongorestore -d cinema -c films movieDetails.bson
2022-01-20T12:36:33.579+0100    checking for collection data in movieDetails.bson
2022-01-20T12:36:33.626+0100    restoring cinema.films from movieDetails.bson
2022-01-20T12:36:33.683+0100    finished restoring cinema.films (2295 documents, 0 failures)
2022-01-20T12:36:33.683+0100    2295 document(s) restored successfully. 0 document(s) failed to restore.
```

L'utilitaire **mongorestore** restaure une sauvegarde binaire créée par **mongodump**. Par défaut, **mongorestore** recherche une sauvegarde de la base de données dans le répertoire dump/. L'utilitaire **mongorestore** restaure les données en se connectant directement à un **mongod** en cours d'exécution. **mongorestore** peut restaurer soit une sauvegarde complète de la base de données, soit un sous-ensemble de la sauvegarde.

On se connecte à **MongoDB** à l'aide de la ligne de commande **$ mongo**

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ mongo
MongoDB shell version v5.0.5
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("2d939833-1236-49c5-9c1a-2718bcf4e586") }
MongoDB server version: 5.0.5
================
```

Pour tester la base de données, on exécute les commandes suivantes :

```
---
> use cinema;
switched to db cinema
> db.films.count()
2295
> db.films.findOne()
{
        "_id" : ObjectId("569190ca24de1e0ce2dfcd4f"),
        "title" : "Once Upon a Time in the West",
        "year" : 1968,
        "rated" : "PG-13",
        "released" : ISODate("1968-12-21T05:00:00Z"),
        "runtime" : 175,
        "countries" : [
                "Italy",
                "USA",
                "Spain"
        ],
```

La fonction **.count()** renvoie le nombre de documents qui correspondraient à une requête .**find()** pour la collection. La méthode .**count()** n'exécute pas l'opération .**find()** mais compte et renvoie le nombre de résultats correspondant à une requête.

Gestion des indexes :

Affichage des indexes :

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

Les fonctions **getIndexes()** afficheront tous les index disponibles pour une collection.

Création d'indexes :

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

Remarque : La fonction **ensureIndex()** n'existe plus dans la version 5.0, il est remplacé par **createIndex()**

```
> db.films.createIndex({"genre": 1}, {"sparse": true})
{
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "createdCollectionAutomatically" : false,
        "ok" : 1
}
>
```

Les opérations **db.collection.insert()** et **db.collection.createIndex()** créent leurs collection si elles n'existent pas déjà.

Affichage des indexes :

```
> db.films.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_"
        },
        {
                "v" : 2,
                "key" : {
                        "genre" : 1
                },
                "name" : "genre_1",
                "sparse" : true
        }
]
>
```

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()**

On exécute donc :

```
> db.films.dropIndex("genre_1")
{ "nIndexesWas" : 2, "ok" : 1 }
>
```

Remarque :
Si le nom de l'index est connu,
> db.collection.dropIndex('name_of_index');
Si le nom de l'index n'est pas connu,
> db.collection.dropIndex({ 'nom_du_champ' : -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().**

L'opérateur **$explain** fournit des informations sur le plan de requête. Il renvoie un document qui décrit le processus et les index utilisés pour renvoyer la requête. Cela peut fournir des informations utiles lors de la tentative d'optimisation d'une requête. Pour plus de détails sur la sortie, voir curseur.

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()
{
        "explainVersion" : "1",
        "queryPlanner" : {
                "namespace" : "cinema.films",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "actors" : {
                                "$eq" : "Bruce Willis"
                        }
                },
                "queryHash" : "592671A4",
                "planCacheKey" : "70938073",
                "maxIndexedOrSolutionsReached" : false,
                "maxIndexedAndSolutionsReached" : false,
                "maxScansToExplodeReached" : false,
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "actors" : {
```

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 :

Si nous les voyons via **.find()** , ils auront l'air très laids :

```
> db.films.find({"year":2012})
{ "_id" : ObjectId("569190cc24de1e0ce2dfcd58"), "title" : "West of Memphis", "year" : 2012, "rated" : "R", "relea
ed" : ISODate("2012-11-22T05:00:00Z"), "runtime" : 147, "countries" : [ "New Zealand", "USA" ], "genres" : [ "Doc
mentary" ], "director" : "Amy Berg", "writers" : [ "Amy Berg", "Billy McMillin" ], "actors" : [ "Michael Baden",
Jason Baldwin", "Holly Ballard", "Jamie Clark Ballard" ], "plot" : "An examination of a failure of justice in the
case against the West Memphis Three.", "poster" : "http://ia.media-imdb.com/images/M/MV5BMjIzNDM3NjkzOV5BMl5BanBn
kFtZTcwNjI5Nzg0OA@@._V1_SX300.jpg", "imdb" : { "id" : "tt2130321", "rating" : 7.9, "votes" : 6627 }, "tomato" : {
"meter" : 95, "image" : "certified", "rating" : 7.9, "reviews" : 111, "fresh" : 106, "consensus" : "Both a soberi
```

Pour contourner ce problème et les rendre lisibles, utilisez la fonction **.pretty()** :

```
> db.films.find({"year":2012}).pretty()
{
        "_id" : ObjectId("569190cc24de1e0ce2dfcd58"),
        "title" : "West of Memphis",
        "year" : 2012,
        "rated" : "R",
        "released" : ISODate("2012-11-22T05:00:00Z"),
        "runtime" : 147,
        "countries" : [
                "New Zealand",
                "USA"
        ],
        "genres" : [
                "Documentary"
        ],
        "director" : "Amy Berg",
        "writers" : [
                "Amy Berg",
                "Billy McMillin"
        ],
        "actors" : [
```

C'est mieux mais ce n'est pas encore parfait car le nombre de documents retournés est important et chaque document est très riche.

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}).pretty()
{
        "_id" : ObjectId("569190cc24de1e0ce2dfcd58"),
        "title" : "West of Memphis",
        "year" : 2012,
        "actors" : [
                "Michael Baden",
                "Jason Baldwin",
                "Holly Ballard",
                "Jamie Clark Ballard"
        ]
}
{
        "_id" : ObjectId("5692a13e24de1e0ce2dfcec7"),
        "title" : "HOUBA! On the Trail of the Marsupilami",
        "year" : 2012,
        "actors" : [
                "Jamel Debbouze",
                "Alain Chabat",
                "Fred Testot",
```

Pour conserver un champ, il suffit de le préciser dans ce document et de lui affecter la valeur 1.

On peut noter que la requête a renvoyé également le **champ `_id`**. C'est le fonctionnement normal dela 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}).pretty()
{
        "title" : "West of Memphis",
        "year" : 2012,
        "actors" : [
                "Michael Baden",
                "Jason Baldwin",
                "Holly Ballard",
                "Jamie Clark Ballard"
        ]
}
{
        "title" : "HOUBA! On the Trail of the Marsupilami",
```

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}).pretty()
{
        "title" : "Shutter Island",
        "year" : 2010,
        "actors" : [
                "Leonardo DiCaprio",
                "Mark Ruffalo",
                "Ben Kingsley",
                "Max von Sydow"
        ]
}
{
        "title" : "Catch Me If You Can",
        "year" : 2002,
        "actors" : [
```

On peut combiner plusieurs filtres : (**.pretty()** par exemple)

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

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}).pre
{
        "title" : "Shutter Island",
        "year" : 2010,
        "actors" : [
                "Leonardo DiCaprio",
                "Mark Ruffalo",
                "Ben Kingsley",
                "Max von Sydow"
        ]
}
{
        "title" : "Big",
        "year" : 1988,
```

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}).pretty()

{
        "title" : "Catch Me If You Can",
        "year" : 2002,
        "actors" : [
                "Leonardo DiCaprio",
                "Tom Hanks",
                "Christopher Walken",
                "Martin Sheen"
        ]
}
>
```

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, nousavons le document **`awards`** :

```
> db.films.findOne({},{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."
        }
}
>
```

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

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 **(>, >=, <, <=, !=)**

Liste des opérateurs de comparaison dans **MongoDB** : **$cmp, $eq, $gt, $gte, $lt, $lte et $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}).pretty()
{
        "title" : "Once Upon a Time in the West",
        "year" : 1968,
        "awards" : {
                "wins" : 4,
                "nominations" : 5,
                "text" : "4 wins & 5 nominations."
        }
}
{
        "title" : "Wild Wild West",
```

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}).pretty()
{
        "title" : "Beasts of the Southern Wild",
        "year" : 2012,
        "awards" : {
                "wins" : 91,
                "nominations" : 119,
                "text" : "Nominated for 4 Oscars. Another 91 wins & 119 nominations."
        }
}
```

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.

Semblable aux méthodes d'agrégation, également par la méthode **.find()**, on a la possibilité de limiter, ignorer, trier et compter les résultats.

```
> db.films.find({"awards.wins":{$gte:80}}, {title:1,year:1,awards:1,_id:0}).sort({"awards.wins":-1}).pretty()
{
        "title" : "No Country for Old Men",
        "year" : 2007,
        "awards" : {
                "wins" : 148,
                "nominations" : 122,
                "text" : "Won 4 Oscars. Another 148 wins & 122 nominations."
        }
}
```

Insértion des documents dans MongoDB :

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

```
> use test ;
switched to db test
> db.test.insertOne({"name":"isbaine_labriji"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("61e94f65d6ec377e4f4935fa")
}
```

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()
{ "_id" : ObjectId("61e94f65d6ec377e4f4935fa"), "name" : "isbaine_labriji" }
```

Si l'on refait la même insertion, on obtient le résultat suivant :

```
> db.test.insertOne({"name":"zaidouni"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("61e94fd7d6ec377e4f4935fb")
}
> db.test.find()
{ "_id" : ObjectId("61e94f65d6ec377e4f4935fa"), "name" : "isbaine_labriji" }
{ "_id" : ObjectId("61e94fd7d6ec377e4f4935fb"), "name" : "zaidouni" }
```

MongoDB a inséré un deuxième document. Le système n'a pas détecté de doublon : il a généré uneclé 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":"isbaine_labriji"})
{ "acknowledged" : true, "insertedId" : 0 }
```

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":"mohammed_saad"})
WriteError({
        "index" : 0,
        "code" : 11000,
        "errmsg" : "E11000 duplicate key error collection: test.test index: _id_ dup key: { _id: 0.0 }",
        "op" : {
                "_id" : 0,
                "name" : "mohammed_saad"
        }
}) :
```

On peut également insérer un document avec une structure différente dans la même collection :

```
> db.test.insertOne({"name":"toto","age":35})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("61e95144d6ec377e4f4935fc")
}
> db.test.find()
{ "_id" : ObjectId("61e94f65d6ec377e4f4935fa"), "name" : "isbaine_labriji" }
{ "_id" : ObjectId("61e94fd7d6ec377e4f4935fb"), "name" : "zaidouni" }
{ "_id" : 0, "name" : "isbaine_labriji" }
{ "_id" : ObjectId("61e95144d6ec377e4f4935fc"), "name" : "toto", "age" : 35 }
>
```

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

```
> db.test.insert([{"name":"mohamed"},{"name": "othman", "ville": ["casablanca","rabat"]}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
```

```
> db.test.find()
{ "_id" : ObjectId("61e94f65d6ec377e4f4935fa"), "name" : "isbaine_labriji" }
{ "_id" : ObjectId("61e94fd7d6ec377e4f4935fb"), "name" : "zaidouni" }
{ "_id" : 0, "name" : "isbaine_labriji" }
{ "_id" : ObjectId("61e95144d6ec377e4f4935fc"), "name" : "toto", "age" : 35 }
{ "_id" : ObjectId("61e9519ad6ec377e4f4935fd"), "name" : "mohamed" }
{ "_id" : ObjectId("61e9519ad6ec377e4f4935fe"), "name" : "othman", "ville" : [ "casablanca", "rabat" ] }
>
```

Il existe déjà un index pour la collecte des transactions. C'est parce que MongoDB crée un index unique sur le **_id** champ lors de la création d'une collection. L'index **_id** empêche les clients d'insérer deux documents avec la même valeur pour le champ **_id**. Vous ne pouvez pas déposer cet index sur le champ **_id.results**.

Supprimer des documents dans MongoDB :

db.collection.drop() de **MongoDB** est utilisé pour supprimer une collection de la base de données.

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

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

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

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

```
> db.test.insertOne({"name":"zaidouni"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("61e952cad6ec377e4f4935ff")
}
> db.test.insertOne({"name":"zaidouni"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("61e952d7d6ec377e4f493600")
}
>
```

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()`.
On tape donc :

```
> db.test.find({"_id":ObjectId("61e952cad6ec377e4f4935ff")})
{ "_id" : ObjectId("61e952cad6ec377e4f4935ff"), "name" : "zaidouni" }
```

Puis :

```
> db.test.deleteOne({"_id":ObjectId("61e952cad6ec377e4f4935ff")})
{ "acknowledged" : true, "deletedCount" : 1 }
>
```

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.

Ajouter ou remplacer un champ existant avec $*set* :

```
> db.test.update({name:"zaidouni"},{$set:{ville:"rabat"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "rabat" }
```

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

Rajoutons ces documents dans notre collection :

```
> db.test.insert({name:"mohammed_isbaine",age:21})
WriteResult({ "nInserted" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "rabat" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 21 }
>
```

Puis :

```
> db.test.insert({name:"zaidouni"})
WriteResult({ "nInserted" : 1 })
> db.test.update({name:"zaidouni"},{$set:{ville:"casablanca"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 21 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni" }
>
```

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. C'est une protection dans **MongoDB** pour empêcher par défaut la mise à jour sur de multiples documents.

Les méthodes **.update()** et **.save()** de **MongoDB** sont utilisées pour mettre à jour le document dans une collection. La méthode **.update()** met à jour les valeurs dans le document existant tandis que la méthode save() remplace le document existant par le document passé dans la méthode **.save()**

Si on souhaite modifier tous les documents, il faut rajouter une instruction **`multi:true`** dans notre fonction **`update`**.

```
> db.test.update({name:"zaidouni"},{$set:{ville:"casablanca"}}, {multi:true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 21 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni", "ville" : "casablanca" }
>
```

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 **« mohammed_isbaine** », nous pouvons exécuterla commande suivante :

```
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 21 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni", "ville" : "casablanca" }
>
>
> db.test.update({name:"mohammed_isbaine"},{$inc:{age:1} })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 22 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni", "ville" : "casablanca" }
>
```

Dans MongoDB, l'opérateur **$*inc*** est utilisé pour incrémenter la valeur d'un champ d'un montant spécifié. L'opérateur **$*inc*** s'ajoute en tant que nouveau champ lorsque le champ spécifié n'existe pas et définit le champ sur le montant spécifié. Le **$*inc*** accepte les valeurs positives et négatives comme montant incrémentiel.

Mettre à jour un tableau avec $*push* ou $*pull* :

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

```
> db.test.insert({"name": "saad_labriji", "ville": ["el_hajeb","rabat"]})
WriteResult({ "nInserted" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 22 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef2483ef5fdc43ad112f7"), "name" : "saad_labriji", "ville" : [ "el_hajeb", "rabat" ] }
> db.test.update({"name": "saad_labriji"}, {$push:{ville:"tanger"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 22 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef2483ef5fdc43ad112f7"), "name" : "saad_labriji", "ville" : [ "el_hajeb", "rabat", "tanger" ] }
>
```

$*push* et **$*pull*** sont les deux opérations utilisées pour ajouter et supprimer des éléments des tableaux dans les documents MongoDB. Les éléments push et pull sont effectués à l'aide des opérateurs **MongoDB** $push et **$*pull***, respectivement : ... L'opérateur **$*pull*** supprime les valeurs d'un tableau qui correspond à une condition spécifiée.

Dans cet exemple, nous avons rajouté **« tanger »** dans le tableau contenant déjà **« el_hajeb »** 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.

$*addToSet* renvoie un tableau de toutes les valeurs uniques résultant de l'application d'une expression à chaque document d'un groupe. L'ordre des éléments dans le tableau renvoyé n'est pas spécifié.

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.update({"name": "saad_labriji"}, {$pull:{ville:"rabat"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.test.find()
{ "_id" : ObjectId("61e952d7d6ec377e4f493600"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef06513fea96d09ecea5a"), "name" : "mohammed_isbaine", "age" : 22 }
{ "_id" : ObjectId("61eef0c913fea96d09ecea5b"), "name" : "zaidouni", "ville" : "casablanca" }
{ "_id" : ObjectId("61eef2483ef5fdc43ad112f7"), "name" : "saad_labriji", "ville" : [ "el_hajeb", "tanger" ] }
>
```

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.

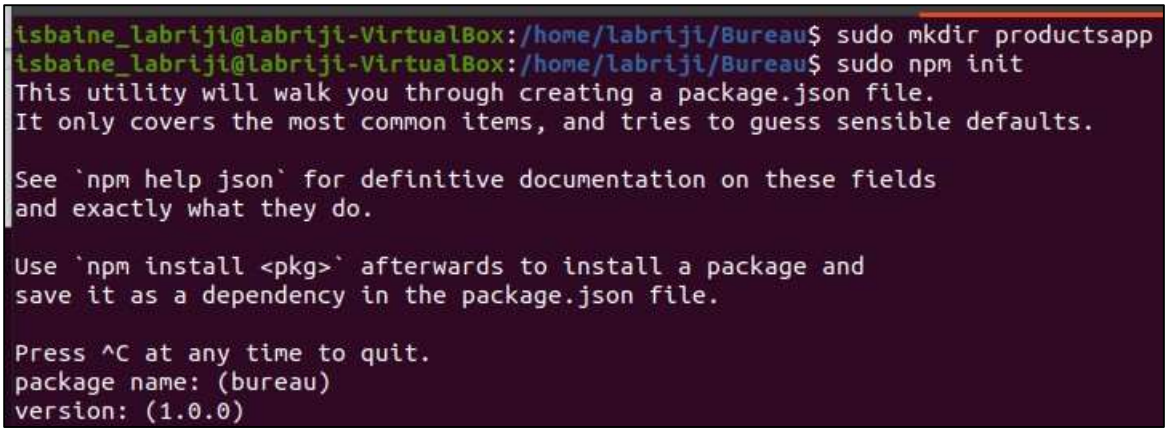
Les étapes d'installation de Node.js :

Installation de la commande npm :

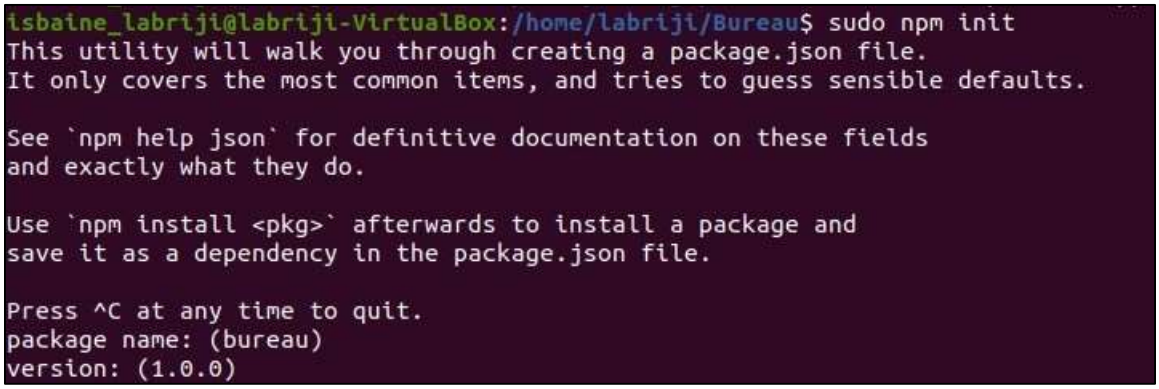
```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Bureau$ sudo apt install npm
[sudo] Mot de passe de isbaine_labriji :
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
```

npm est le gestionnaire de packages pour la plate-forme **Node JavaScript**. Il met en place des modules pour que le nœud puisse les trouver, et gère intelligemment les conflits de dépendances. Il est extrêmement configurable pour prendre en charge une grande variété de cas d'utilisation. Le plus souvent, il est utilisé pour publier, découvrir, installer et développer des programmes de nœud.

Création un répertoire appelé «**productsapp**» dans le répertoire : **/home/labriji/Documents** (par exemple, le Screenshot suivant n'est pas mis à jour !, le dossier qu'on utilisera est **Documents**) :



Dans ce répertoire nouvellement créé, on exécute la commande suivante :



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.

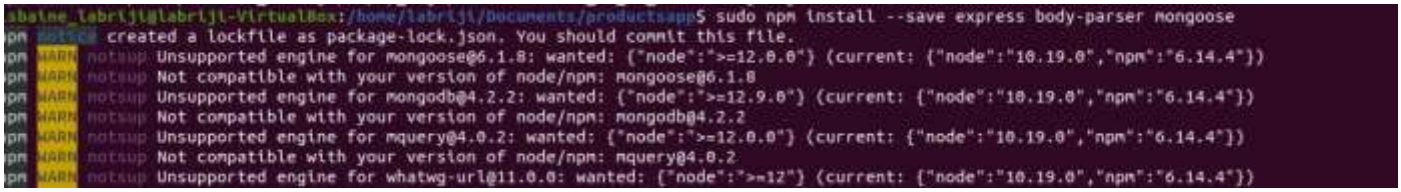
npm init <initializer> peut être utilisé pour configurer un package **npm** nouveau ou existant. **initializer** dans ce cas est un package **npm** nommé **create-<initializer>** , qui sera installé par **npx** , puis son bac principal exécuté - probablement en créant ou en mettant à jour le package.

Installation des packages nécessaires :

Nous allons installer les packages que nous utiliserons pour notre **API** qui sont :

- **ExpressJS** : c'est une application Web Node.JS flexible qui a de nombreuses fonctionnalités pour les applications Web et mobiles.
- **mongoose** : c'est la librairie d'Object Data Modeling (**ODM**) pour MongoDB and Node.js.
- **body-parser** : package pouvant être utilisé pour gérer les requêtes **JSON**.

Nous pouvons installer les packages mentionnés ci-dessus en tapant ce qui suit.



Initialisation du serveur :

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

On ouvre le nouveau fichier nommé **app.js**, puis on ajoute toutes les dépendances précédemment installées (**ExpressJS et body-parser**) en insérant le contenu suivant dans :

```
1 // app.js
2 const express = require('express');
3 const bodyParser = require('body-parser');
4 // initialize our express app
5 const app = express();
6
```

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

```
 8 let port = 1234;
 9 app.listen(port, () => {
10 console.log('Server is up and running on port number ' + port);
11 });
```

Maintenant, nous testons notre serveur en utilisant la commande suivante dans le terminal :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
Server is up and running on port number 1234
```

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 :

MVC est un modèle de conception utilisé pour découpler l'interface utilisateur (**vue**), les données (**modèle**) et la logique d'application (**contrôleur**). Ce modèle aide à réaliser la séparation des préoccupations.

En utilisant le **modèle** MVC pour les sites Web, les demandes sont acheminées vers un contrôleur chargé de travailler avec le modèle pour effectuer des actions et/ou récupérer des données. Le contrôleur choisit la vue à afficher et lui fournit le modèle. La vue affiche la page finale, en fonction des données du modèle.

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 codes 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 :

- **controllers**
- **models**
- **routes**
- **views**

Pour cela, on tape les commandes suivantes :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo mkdir controllers
[sudo] Mot de passe de isbaine_labriji :
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo mkdir models
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo mkdir routes
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo mkdir views
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$
```

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. On crée un nouveau fichier dans le répertoire « **models** » et appelons-le « **product.model.js** » :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ cd models/
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp/models$ sudo vi product.model.js
```

```
1 const mongoose = require('mongoose');
2 const Schema = mongoose.Schema;
3 let ProductSchema = new Schema({
4 name: {type: String, required: true, max: 100},
5 price: {type: Number, required: true},
6 });
7 // Export the model
8 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, on crée un fichier « **product.route.js** ». Il s'agit du fichier qui contiendra lesroutes des produits.

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ cd routes/
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp/routes$ sudo vi product.route.js
```

On copie le contenu suivant dans **product.route.js** :

```
1 const express = require('express');
2 const router = express.Router();
3 // Require the controllers WHICH WE DID NOT CREATE YET!!
4 const product_controller = require('../controllers/product.controller');
5 // a simple test url to check that all of our files are communicating correctly.
6 router.get('/test', product_controller.test);
7 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**.

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ cd controllers/
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp/controllers$ sudo vi product.controller.js
```

```
1 const Product = require('../models/product.model');
2 //Simple version, without validation or sanitation
3 exports.test = function (req, res) {
4 res.send('Greetings from the Test controller!');
5 };
```

La dernière étape avant d'essayer notre première route consiste à ajouter la classe route à **app.js** :

```
1 // app.js
2 const express = require('express');
3 const bodyParser = require('body-parser');
4 const product = require('./routes/product.route'); // Imports routes for the
5 products
6 const app = express();
7 app.use('/products', product);
8 const port = 1234;
9 app.listen(port, () => {
10 console.log('Server is up and running on port number ' + port);
11 });
```

On exécute la commande suivante : (pour la version récente de mongo (v5+), il y a une erreur qui apparaît)

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
/home/labriji/Documents/productsapp/node_modules/whatwg-url/lib/encoding.js:2
const utf8Encoder = new TextEncoder();
                    ^

ReferenceError: TextEncoder is not defined
    at Object.<anonymous> (/home/labriji/Documents/productsapp/node_modules/whatwg-url/lib/encoding.js:2:21)
    at Module._compile (internal/modules/cjs/loader.js:778:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
```

Pour fixer l'erreur, il faut faire les modifications suivantes :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ cd node_modules/whatwg-url/lib/
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp/node_modules/whatwg-url/lib$ sudo vi encoding.js
```

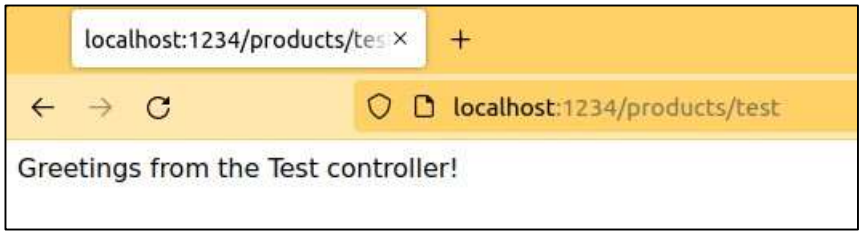
Il faut commenter la 2éme et la 3éme ligne et à leur place, ajoutez les 3 lignes suivantes :
→ *var util= require('util');*
→ *const utf8Encoder = new util.TextEncoder();*
→ *const utf8Decoder = new util.TextDecoder("utf-8", { ignoreBOM: true });*

```
"use strict";
//const utf8Encoder = new TextEncoder();
//const utf8Decoder = new TextDecoder("utf-8", { ignoreBOM: true });
var util= require('util');
const utf8Encoder = new util.TextEncoder();
const utf8Decoder = new util.TextDecoder("utf-8", { ignoreBOM: true });

function utf8Encode(string) {
  return utf8Encoder.encode(string);
}

function utf8DecodeWithoutBOM(bytes) {
  return utf8Decoder.decode(bytes);
}

module.exports = {
  utf8Encode,
  utf8DecodeWithoutBOM
};
```

Enfin, on tape la commande :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
[sudo] Mot de passe de isbaine_labriji :
Server is up and running on port number 1234
```

Puis on se dirige vers notre navigateur et on essaye le lien suivant :
→ *http://localhost:1234/products/test*



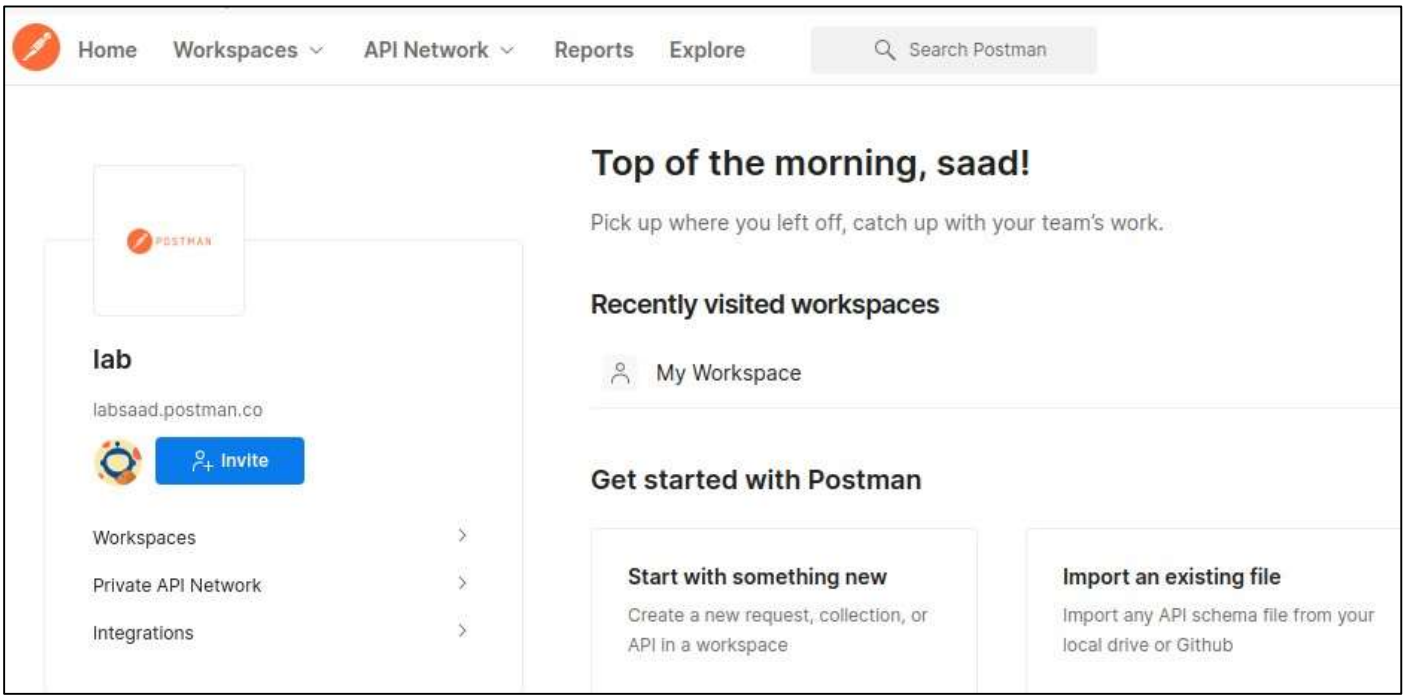
Postman :

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 **Endpoint** que nous allons implémenter dans ce **TP**. Mais tout d'abord, nous allons installer et tester **Postman**.

Installation de Postman :
On ouvre un nouveau terminal et on tape :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo snap install postman
[sudo] Mot de passe de isbaine_labriji :
Télécharger un paquet Snap "postman" (166) à partir du canal "v9/stable"                46% 10.7MB/s 10.0s
```

Après l'installation, on tape **« postman »** pour pouvoir l'utiliser : **$ postman** ,nous pouvons nous connecter avec notre compte google avec **« SignIn »** .

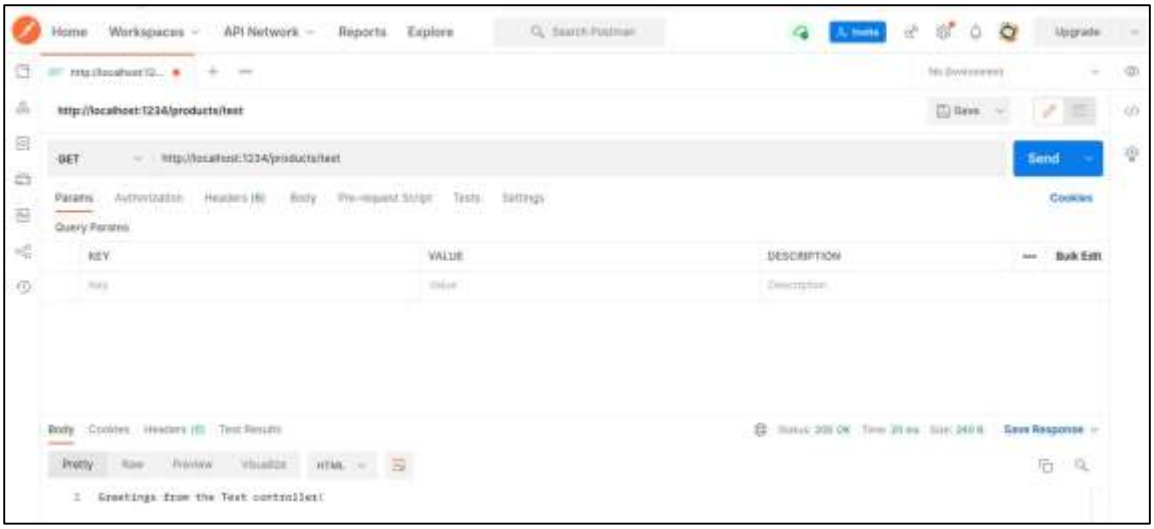


Ensuite, on clique sur **« create a request»**. Puis, on choisis **« GET »** comme requête et on colle l'url suivant : → **localhost:*1234/products/test*** puis on clique sur **« SEND ».**

Nous devons nous assurer d'abord que votre serveur fonctionne toujours sur le port numéro **1234**.

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
[sudo] Mot de passe de isbaine_labriji :
Server is up and running on port number 1234
```

On peut 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 installer 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);**

```
12 // Set up mongoose connection
13 var mongoose = require('mongoose');
14 var dev_db_url = 'mongodb://127.0.0.1/tp_database';
15 var mongoDB = process.env.MONGODB_URI || dev_db_url;
16 mongoose.connect(mongoDB, { useNewUrlParser: true });
17 mongoose.Promise = global.Promise;
18 var db = mongoose.connection;
19 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, on ajoute les deux lignes suivantes avant la ligne : **app.use('/products', product);**

```
5 const app = express();
6 app.use(bodyParser.json());
7 app.use(bodyParser.urlencoded({extended: false}));
8 app.use('/products', product);
```

Voici à quoi ressemble notre fichier **app.js** complet :

```
// app.js

var express = require('express');
var bodyParser = require('body-parser');

var product = require('./routes/product.route'); // Imports routes for the productsvar 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);var port = 1234;


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

On tape la commande suivante pour démarrer notre application :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
[sudo] Mot de passe de isbaine_labriji :
Server is up and running on port number 1234
```

Implémentation des Endpoint :

CREATE :

La première tâche de nos opérations **CRUD** est de créer un nouveau produit. On commence par définir d'abord notre route. On se dirige vers products.route.js et on commence à concevoir le chemin attendu que le navigateur atteindrait et le contrôleur qui serait responsable de la gestion de cette demande.

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

```
6 router.get('/test', product_controller.test);
7 router.post('/create', product_controller.product_create);
8 module.exports = router;
```

On écrit maintenant le contrôleur **product_create** dans notre fichier « **product.controller.js** ».

On ajoute les lignes suivantes à la fin du fichier « **product.controller.js** ».

```
6 exports.product_create = function (req, res, next) {
7 let product = new Product(
8 {
9 name: req.body.name,
10 price: req.body.price
11 }
12 );
13 product.save(function (err) {
14 if (err) {
15 return next(err);
16 }
17 res.send('Product Created successfully')
18 })
19 };
```

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.

On tape la commande pour prendre en compte les modifications :

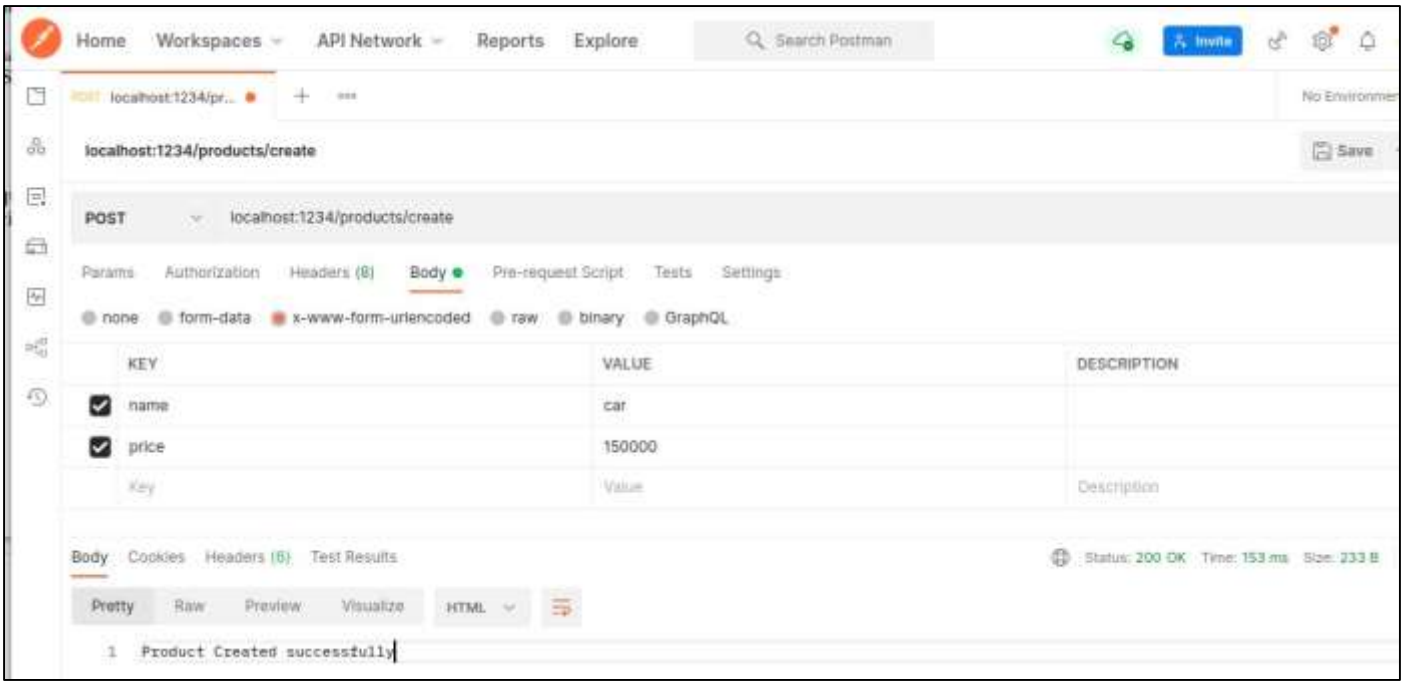
```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
[sudo] Mot de passe de isbaine_labriji :
Server is up and running on port number 1234
```

On ouvre **Postman**, en tapant la commande : **`$postman`**

Envoyons ensuite une demande POST à l'URL suivante : ***localhost:1234/products/create*** et on spécifie les données POST, par exemple :

name: **car** et *price:* **150000**

On s'assure également qu'on a choisis : **x-www-form-urlencoded** dans l'onglet Body tab dans Postman comme spécifié dans l'image ci-dessous :



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 **« car »** 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, on tape : **`$mongo`**

```
labriji@labriji-VirtualBox:~/Bureau$ sudo mongo
[sudo] Mot de passe de labriji :
```

Puis :

```
> show dbs
admin        0.000GB
cinema       0.001GB
config       0.000GB
local        0.000GB
mydb         0.000GB
test         0.000GB
tp_database  0.000GB
> use tp_database
switched to db tp_database
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 150000, "__v" : 0 }
>
```

On remarque le nouvel produit crée a été ajouter a notre base de données.

Read :

La deuxième tâche de notre application CRUD consiste à lire un produit existant. Pour configurer la route :
On ajouter cette ligne dans **products.route.js**, juste avant la ligne : **module.exports = router;** :

```
8 router.get('/:id', product_controller.product_details);
9 module.exports = router;
```

On écrit maintenant le contrôleur **product_details** dans notre fichier contrôleur. On ajoute les lignes suivantes à la fin du fichier « **product.controller.js** » :

```
21 exports.product_details = function (req, res, next) {
22 Product.findById(req.params.id, function (err, product)
23 {
24 if (err) return next(err);
25 res.send(product);
26 })
27 };
```

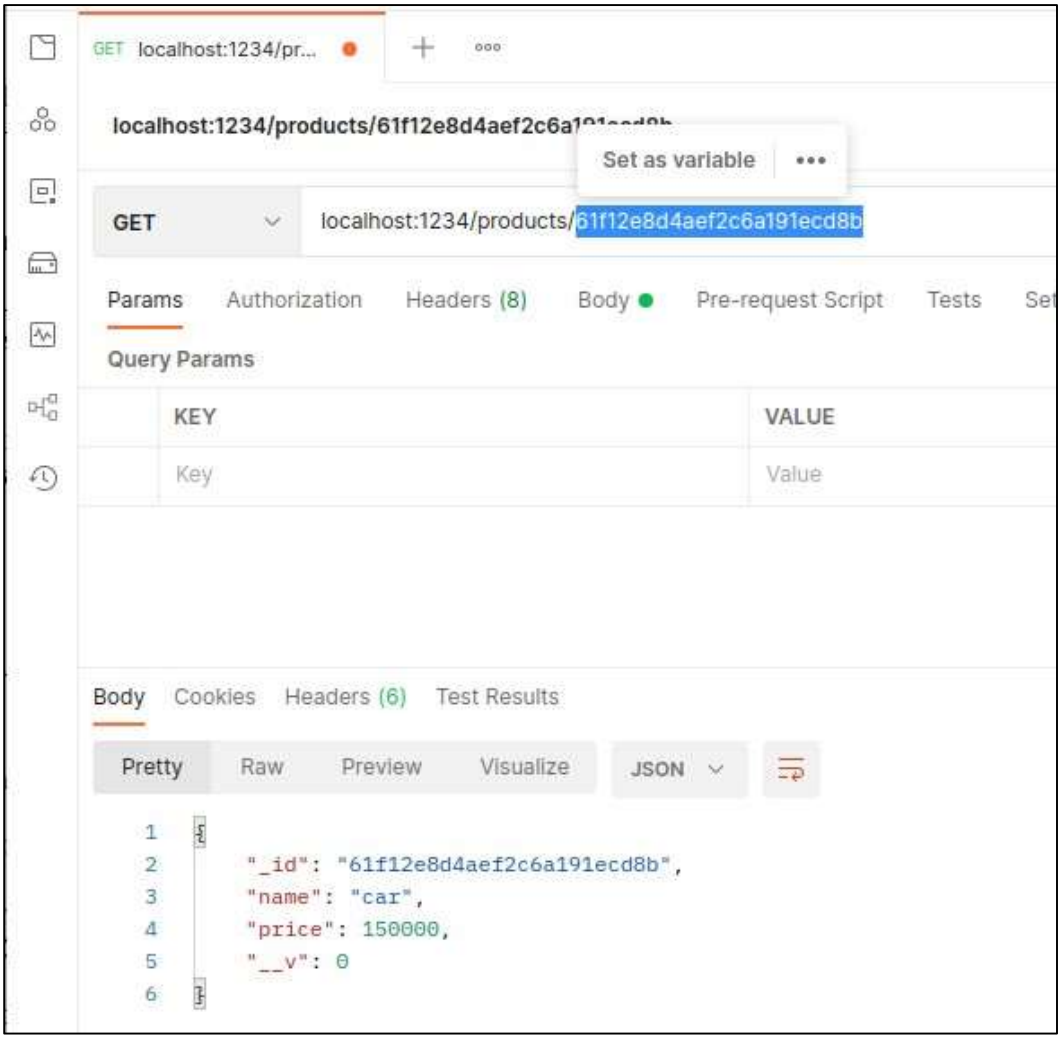
La fonction consiste simplement à lire un produit existant à partir de l'ID de produit envoyé dans la demande. Maintenant, il faut relancer la commande pour prendre en compte les modifications :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
[sudo] Mot de passe de isbaine_labriji :
Server is up and running on port number 1234
```

On ouvre **Postman** et on essaye notre nouveau **Endpoint**. On choisit « **GET** » et on appele l'**URL** suivant : ***localhost:1234/products/PRODUCT_ID***

PRODUCT_ID est l'**ID** de l'objet que nous avons créé dans le Endpoint précédent. On peut l'obtenir de votre base de données :

```
> use tp_database
switched to db tp_database
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 150000, "__v" : 0 }
>
```



Nous avons obtenu une réponse contenant toutes les informations de ce produit spécifique. Vous pouvez voir qu'il s'appelle « **car** » et que son price est « **150000** »

Update :

La troisième tâche de notre application CRUD est de mettre à jour un produit existant.

Pour configurer la route, on ajoute cette ligne dans **products.route.js**, juste avant la ligne : **module.exports = router;** :

```
 8 router.get('/:id', product_controller.product_details);
 9 router.put('/:id/update',
10 product_controller.product_update);
11 module.exports = router;
```

On écrit maintenant le contrôleur **product_update** dans notre fichier contrôleur.

On ajoute les lignes suivantes à la fin du fichier « **product.controller.js** » :

```
27 };
28
29 exports.product_update = function (req, res,next) {
30 Product.findByIdAndUpdate(req.params.id, {$set:req.body}, function (err, product)
31 {
32 if (err) return next(err);
33 res.send('Product udpated.');
34 });
35 };
```

La fonction détecte simplement un produit existant en utilisant son identifiant envoyé dans la demande.

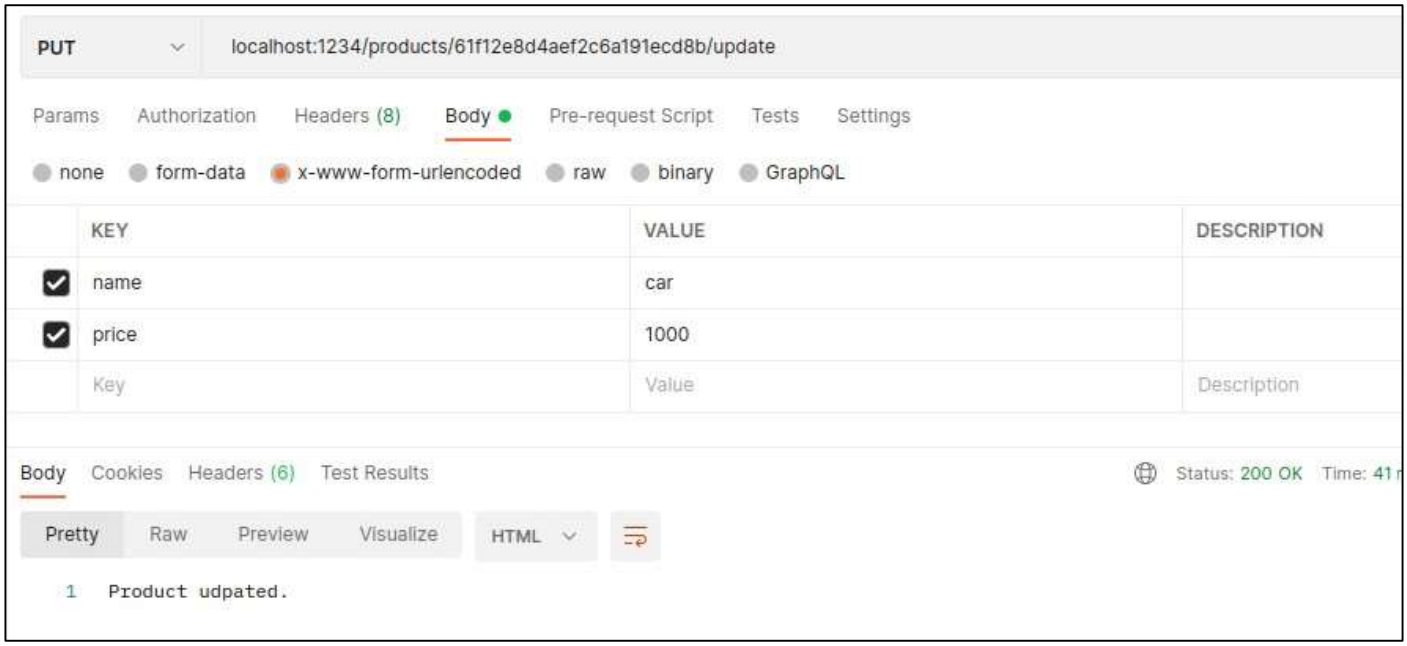
Maintenant, on relance la commande pour prendre en compte les modifications :

```
isbaine_labriji@labriji-VirtualBox:/home/labriji/Documents/productsapp$ sudo node app.js
[sudo] Mot de passe de isbaine_labriji :
Server is up and running on port number 1234
```

On ouvre **Postman** et on essaye notre nouveau **Endpoint**. On choisit « **Put** » et on appele 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. On peut l'obtenir de votre base de données.

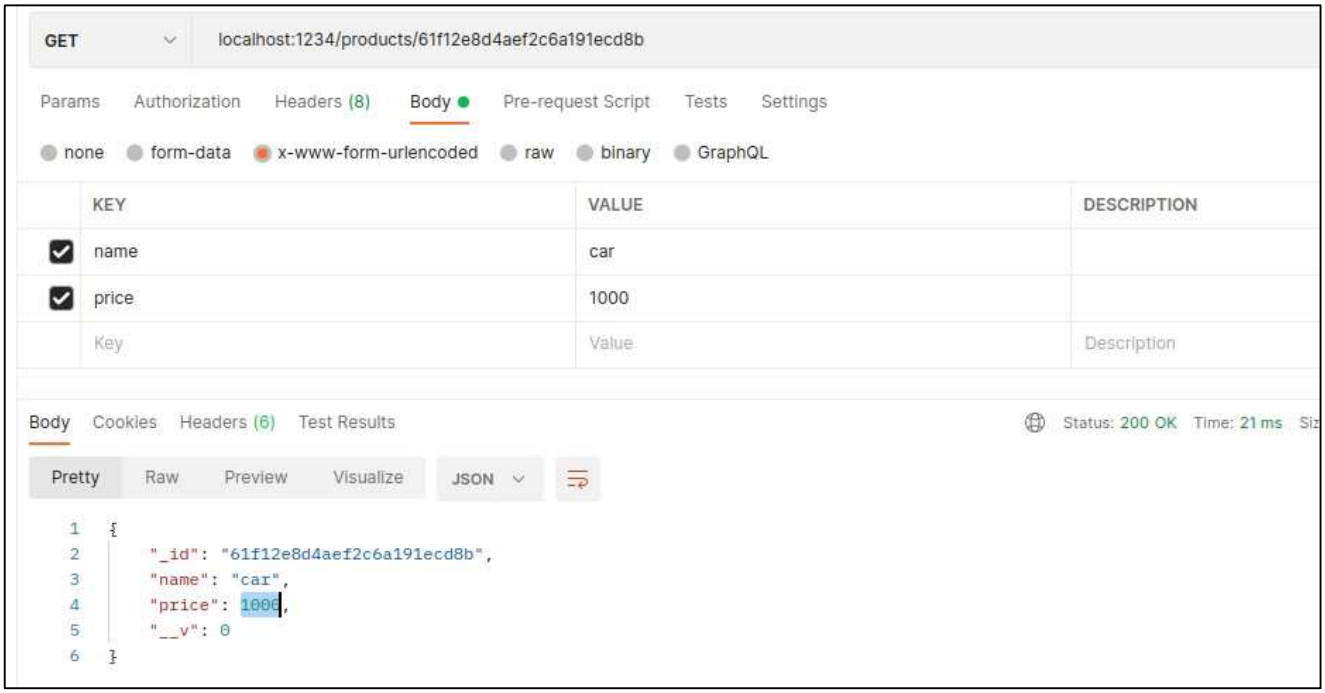
```
> use tp_database
switched to db tp_database
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 150000, "__v" : 0 }
>
```



Pour tester ces modifications, on se connecte au terminal de **Mongo** et on vérifie que la collection **products** a été bien modifiée.

```
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 150000, "__v" : 0 }
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 1000, "__v" : 0 }
>
```

Ou bien : (on peut utiliser **Postman**)



Delete :

La dernière tâche de notre application **CRUD** est de supprimer un produit existant.

Pour configurer la route :
On ajouter cette ligne dans **products.route.js**, juste avant la ligne **: module.exports = router;** :

```
10 product_controller.product_update);
11 router.delete('/:id/delete',
12 product_controller.product_delete);
13 module.exports = router;
```

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

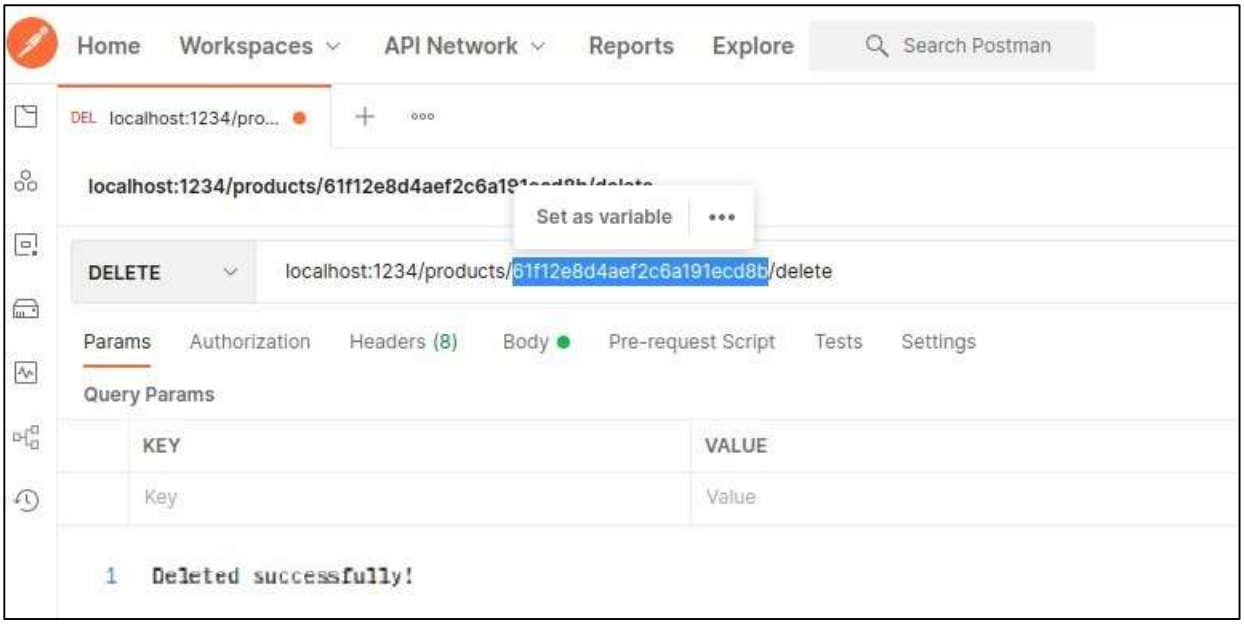
```
36
37 exports.product_delete = function (req, res, next) {
38 Product.findByIdAndRemove(req.params.id, function (err)
39 {
40 if (err) return next(err);
41 res.send('Deleted successfully!');
42 })
43 };
```

La fonction supprime simplement un produit existant.

On ouvre **Postman** et on essaye notre nouveau **Endpoint**. On choisit « **DELETE** » et on appele l'**URL** suivant :
localhost:1234/products/PRODUCT_ID/delete

```
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 150000, "__v" : 0 }
> db.products.find()
{ "_id" : ObjectId("61f12e8d4aef2c6a191ecd8b"), "name" : "car", "price" : 1000, "__v" : 0 }
>
```

PRODUCT_ID est l'**ID** de l'objet que nous avons créé dans le point de terminaison précédent. On peut l'obtenir de notre base de données.



Pour tester ces modifications, on se connecte au terminal de **Mongo** et on vérifiez que la collection **products** a été bien supprimée.

Pour cela taper : **`$mongo`**
Puis :

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

Isbaine Labriji data ine2