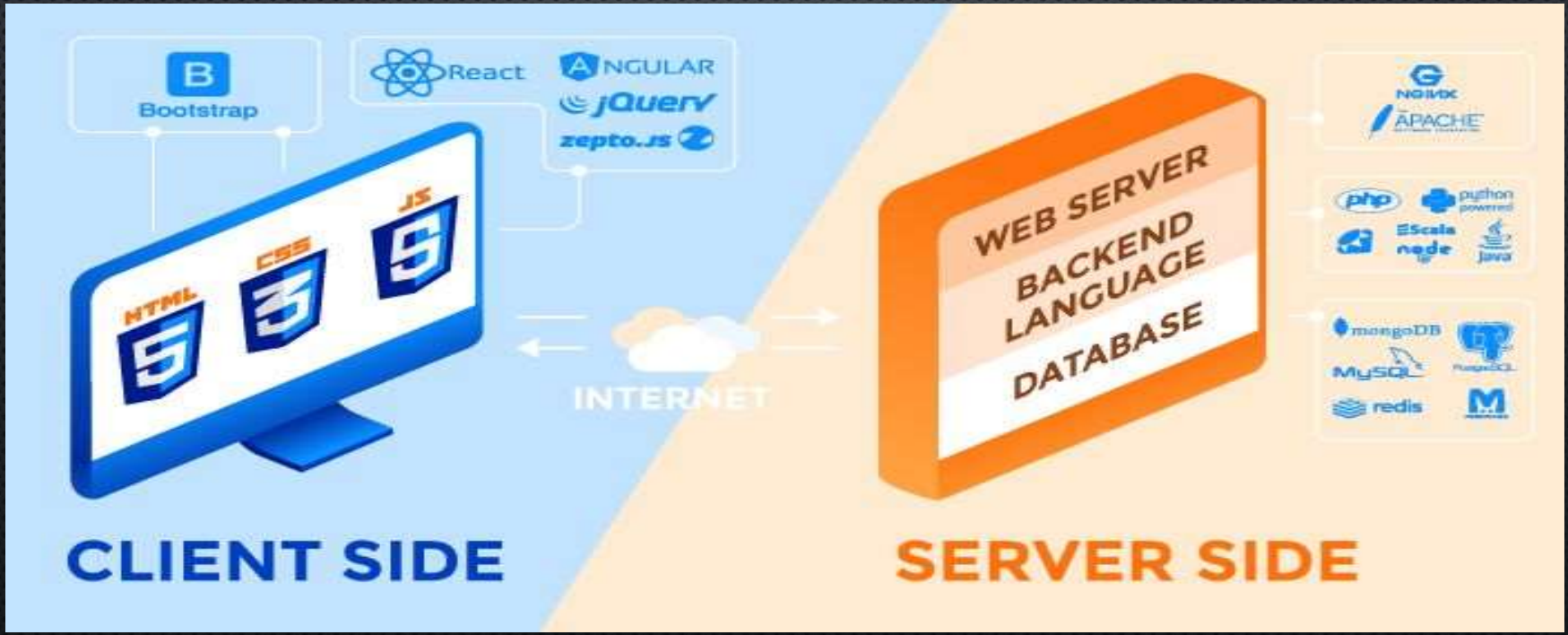
CRÉATION D'UNE API REST AVEC EXPRESS NODEJS

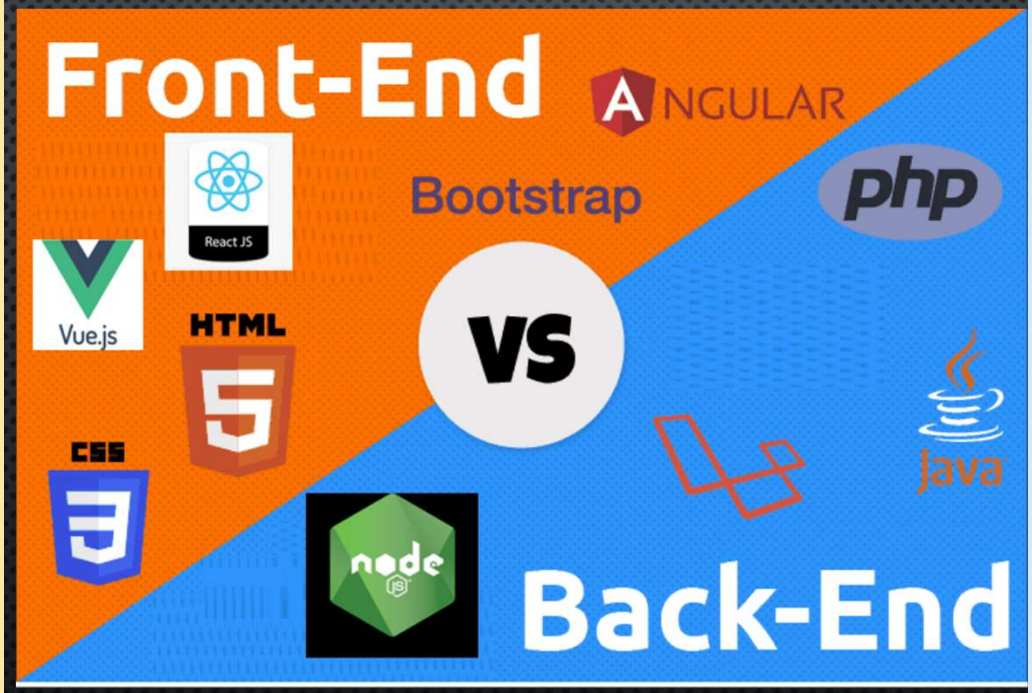
COMPRÉHENSION DES CONCEPTS CLÉS : RÔLE DU CÔTÉ SERVEUR DANS LE DÉVELOPPEMENT WEB



FRONT-END / BACK-END

DIFFÉRENCE ENTRE FRONT-END ET BACK-END

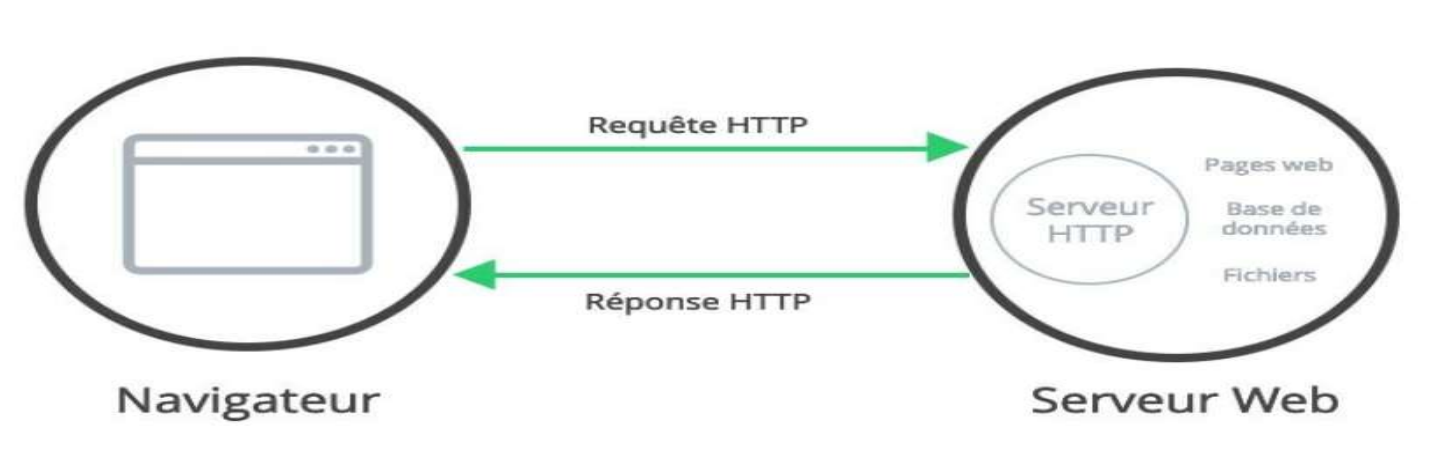
- **Front-End** désigne les éléments d'un site que l'on voit à l'écran et avec lesquels on peut interagir depuis un navigateur.
- Tout ce qu'on voit sur un site internet est une combinaison de HTML, CSS et JavaScript.
- Ces langages de programmation utilisés par le développeur Front-End sont interprétés par le navigateur de l'ordinateur pour afficher un résultat « visuel ».



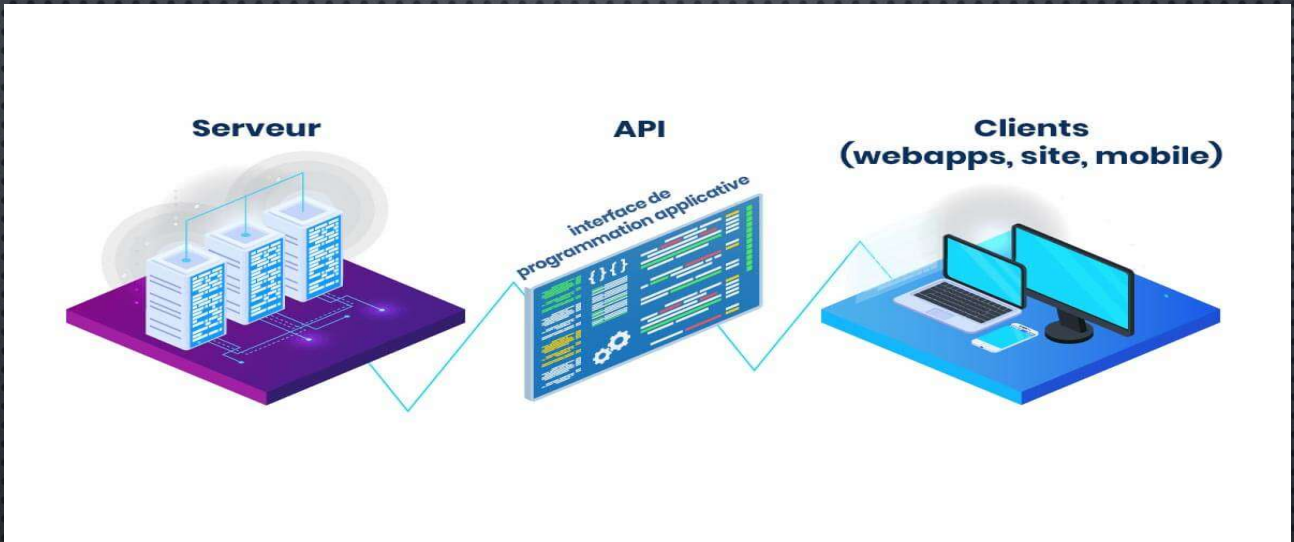
- Le **Back end** se compose généralement d'un serveur, d'une application et d'une base de données
- Les technologies Back end se composent de langages comme NodeJS, PHP, Python, etc.
- Ils sont généralement améliorés par des Frameworks comme Express et Laravel qui rendent le développement plus rapide et plus sécurisé.

QU'EST-CE QU'UN SERVEUR WEB ?

- Un serveur web est un ensemble matériel et logiciel qui permettent l'accès aux fichiers hébergés, page web et base de données stocké sur un ordinateur.
- Le serveur web est constitué en autre d'un serveur HTTP. Le serveur HTTP est un logiciel qui comprend/reçoit les URL et les requêtes via le protocole HTTP (le protocole utilisé par le navigateur pour afficher les pages web).
- Au niveau le plus simple, à chaque fois qu'un navigateur a besoin d'un fichier ou autre hébergé sur un serveur web, le navigateur fait la demande au serveur (on dit qu'il envoie une requête HTTP). Quand la requête atteint le serveur, le serveur HTTP la traite et renvoie le réponse.

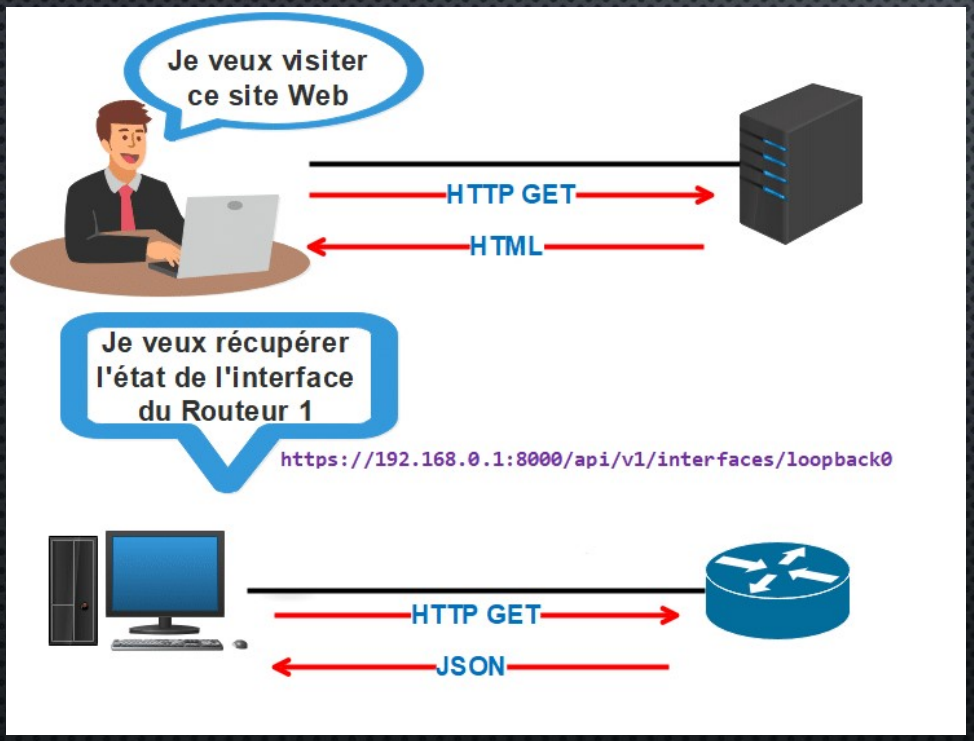


API



- Une API, ou interface de programmation d'application, est un ensemble de définitions et de protocoles qui facilite la création et l'intégration des applications.
- es API permettent à votre produit ou service de communiquer avec d'autres produits et services sans connaître les détails de leur mise en œuvre.

API REST



- Une API REST (REpresentational State Transfer) est une interface de programmation d'application (API ou API web) qui respecte les contraintes du style d'architecture REST et permet d'interagir avec les services web RESTful.
- Ainsi, lorsqu'un utilisateur effectue une requête via une API RESTful, celle-ci transfère une représentation de l'état des ressources requises au système client.
- Cela signifie que les développeurs n'ont pas besoin d'installer des bibliothèques ou des logiciels supplémentaires pour tirer parti d'une conception d'API REST.

REQUÊTES HTTP

- Ce sont les actions que vous souhaitez effectuer sur la ressource indiquée.
- Ces méthodes sont souvent appelées verbes HTTP.

- Voici les verbes/actions HTTP les plus couramment utilisés
  - GET : Les requêtes GET sont utilisées que pour récupérer des données.
  - POST : Les requêtes GET sont utilisées pour envoyer des nouvelles données.
  - PUT : Les requêtes PUT sont utilisées pour modifier des données.
  - PATCH : Les requêtes PATCH sont utilisées pour modifier partiellement des données.
  - DELETE : Les requêtes DELETE supprime les données indiquées.

FRAMEWORK C'EST QUOI ?

- Un **Framework** (cadre de travail en Français) est une infrastructure qui sert de cadre au développement d'une solution (logiciel, application web, site internet) sur mesure. Il intègre un ensemble d'outils permettant de créer, d'ajouter et de développer des fonctionnalités ou des interfaces.
- Le Framework est la solution qui s'utilise généralement pour la conception des sites web dynamiques.
- L'objectif essentiel d'un Framework est d'aider les développeurs dans la réalisation de leurs tâches.
- L'organisation et la structure du Framework vise la productivité maximale du développeur qui va l'utiliser et facilite la maintenance du logiciel.

FRAMEWORK WEB

- Web Development Framework ou simplement Framework Web est un Framework logiciel qui offre un ensemble d'outils pour prendre en charge le développement d'applications Web,
- Notamment des services Web, des ressources Web et des API Web (interfaces de programmation d'applications).
- Le cadre de développement Web offre une large gamme de composants pré-écrits, d'extraits de code et de modèles d'application complets pour faciliter le travail des développeurs Web.

QU'EST-CE QU'UNE PILE ?

- Une "pile" fait référence à toute combinaison de langages de programmation et de technologies ou à une combinaison de produits logiciels.
- La pile complète commence avec la base de données et le serveur Web dans le back-end, contient la logique d'application et le contrôle au milieu, et va jusqu'à l'interface utilisateur du front-end.
- Lorsque nous parlons de développement full-stack, nous parlons de développer toutes les parties d'un site Web ou d'une application.

FULL STACK DEVELOPMENT

- Le développement full stack fait référence au processus de conception, de création et de maintenance de tous les aspects d'une application Web, y compris les composants front end (côté utilisateur) et back end (côté serveur).
- En tant que développeur full stack, vous avez la possibilité de créer une application Web du début à la fin, garantissant une intégration transparente entre l'interface utilisateur et les processus côté serveur.

NODE.JS

QU'EST-CE QUE NODE.JS ?

- Node.js est un environnement serveur, open source, gratuit, et qui utilise le language JavaScript pour créer des applications Web côté serveur. (back end)
- Node.js fonctionne sur différentes plateformes (Windows, Linux, Unix, Mac OS X, etc.)
- Avec Node.js, nous pouvons créer une application Web rapide et hautement évolutive.
- L'utilisation de Node.js signifie également que nous pouvons utiliser JavaScript sur l'ensemble du stack c'est à dire le même langage pour le front end et le back end.
- Donc un développement plus rapide et efficace.
- Node.js a une énorme bibliothèque de packages prêts à l'emploi et qui vous feront gagner beaucoup de temps. Ces bibliothèques sont gérées par NPM (Node Package Manager)
- Une particularité de Node.js est qu'il utilise la programmation asynchrone (exécution de plusieurs tâches en même temps) vs la programmation synchrone (exécution d'une tâche à la fois) que l'on trouve sur de nombreux langages de programmation côté serveur comme PHP et Ruby.

PRINCIPE

- Node.js ne s'agit pas d'un simple langage de script tournant sur un serveur préexistant ; en effet, il faut coder soi-même les différentes couches du serveur.
- Il faudra ainsi mettre en place :
  - le serveur HTTP lui-même, avec la gestion des différents messages HTTP
  - le routage des requêtes vers les bonnes pages du serveur
  - de quoi récupérer les paramètres saisis dans l'URL
  - un système permettant d'envoyer du contenu au client en fonction de l'URL
  - un dispositif permettant, en JavaScript, de gérer des accès au système de fichiers pour permettre non seulement la lecture, mais aussi l'écriture et même l'envoi de fichiers par l'utilisateur.

INSTALLATION DE NODE.JS

- Node.js peut être installé directement à partir de son site Web : https://nodejs.org

- Une fois l'installation terminée, vous pouvez vérifier la version installée avec cette commande

node -v

EXPRESS.JS

C'EST QUOI EXPRESS ?

- Express.js, parfois aussi appelé « Express », est un Framework back end Node.js minimaliste, rapide qui offre des fonctionnalités et des outils robustes pour développer des applications back end évolutives.
- Il vous offre le système de routage et des fonctionnalités simplifiées pour étendre le Framework en développant des composants et des parties plus puissants en fonction des cas d'utilisation de votre application.
- Vous pouvez développer des applications, des points de terminaison API, des systèmes de routage et des Frameworks avec lui.

CRÉATION D'UNE NOUVELLE APPLICATION

- D'abord créer un dossier pour accueillir votre application
- La première étape est d'y créer le fichier package.json.

npm init

- Ensuite, installation du package ExpressJS

npm install express

NODEMON

- Le package nodemon permet de recharger le serveur à chaque modification de notre code.
- Pour l'installer :

npm i -g nodemon

COMMENT FONCTIONNE EXPRESS ?

- Créons un serveur simple pour écouter les requêtes entrantes provenant d'une URL et d'un numéro de port spécifiques avec Express.js.
- Dans le répertoire du projet créé, créez un fichier appelé app.js et ajoutez le code suivant :

```
const express = require('express')
const app = express()
const port = 3001

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Server is listening on port ${port}`)
})
```

- Dans ce code, le app.listen() va créer un serveur HTTP et lire les requêtes entrantes.
- L'application démarre un serveur et écoute le port 3001 à la recherche de connexions. L'application répond "Hello World!" aux demandes adressées à l'URL racine (/) ou à la route racine. Pour tous les autres chemins d'accès, elle répondra par 404 Not Found.

EXÉCUTER L'APPLICATION

- Exécutez l'application avec la commande suivante :

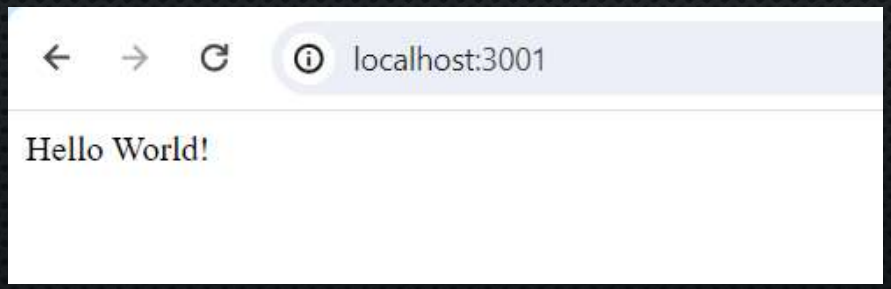
node app.js

Ou puisque nodemon est installé exécuter :

nodemon app

```
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
Server is listening on port 3001
```

- Puis chargez http://localhost:3001/ dans un navigateur pour consulter le résultat.



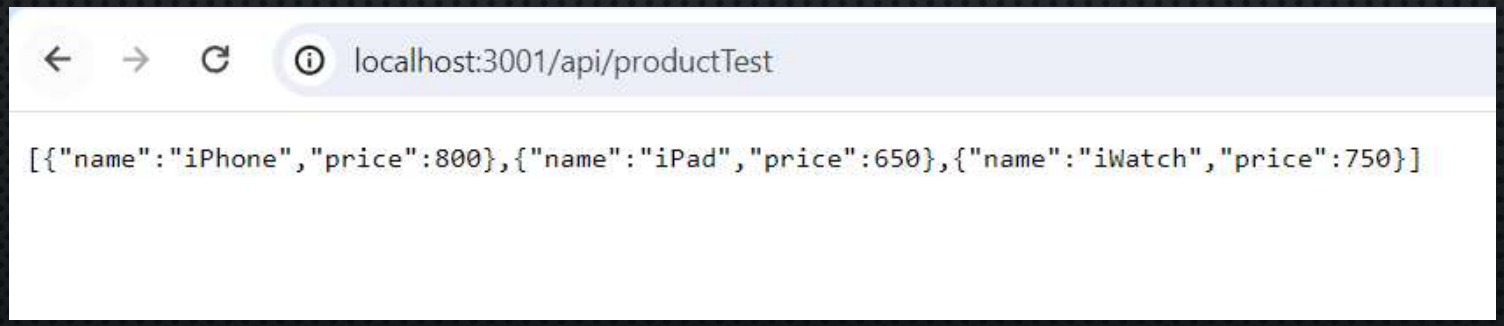
EXEMPLE API

- Modifier le code :

```
const express = require('express')
const app = express()
const port = 3001

app.get('/', (req, res) => {
 res.send('Hello World!')
})
app.get('/api/productTest', (req, res) => {
   res.json([
      { name: 'iPhone', price: 800 },
      { name: 'iPad', price: 650 },
      { name: 'iWatch', price: 750 }
   ])
})

app.listen(port, () => {
 console.log(`Server is listening on port ${port}`)
})
```

TEST DE L'API

- Le app.get() définit un chemin URL qui retourne une liste de produit JSON.
- Accédez à localhost:3001/api/productTest
- Vous devriez voir une représentation JSON de la liste des produits



CRÉATION D'UNE API REST AVEC EXPRESS ET MONGOOSE

MONGODB

- MongoDB est un système de gestion de base de données orienté documents, de la classe des bases de données NoSQL.
- Appelée également « Not Only SQL » (pas seulement SQL), la base de données NoSQL est une approche de la conception des bases et de leur administration particulièrement utile pour de très grands ensembles de données distribuées.
- Dans un système de base de données relationnelles les données sont stockées par ligne dans des tables.
- Dans MongoDB, les données sont modélisées sous forme de documents sous un style JSON.
- On ne parle plus de tables, ni d'enregistrements mais de collections et de documents.

EXEMPLE DE COLLECTION

```
{
    "_id" : ObjectId("54ce14c9dc4d5f155a233301"),
    "nom" : "GoodFellas",
    "acteurs" : [
        {
            "nom" : "Deniro",
            "prenom" : "Robert"
        },
        {
            "nom" : "Liota",
            "prenom" : "Ray"
        }
    ]
}
```

MONGODB, UN SGBD NOSQL

- A la différence des systèmes SQL qui reposent sur des tables relationnelles, MongoDB repose sur des collections de documents au format JSON (clé/valeur).

- Exemple d'une collection contenant deux documents.

COLLECTION

```
[
  {
    "_id": "586a759d687ee42fd410d5bb",
    "description": "La piscine Fives de Lille est une piscine couverte
        apprentissage, perfectionnement pour les enfants et les adulte
        jardin aquatique (enfants de 0 à 4 ans), aquagym seniors ; cent
        natation…",
    "tel": "03 20 56 38 62",
    "adresse": "82 rue du Long Pot, 59000  Lille",
    "nom": "Piscine de Fives",
    "__v": 0
  },
  {
    "_id": "586a76eb687ee42fd410d5be",
    "description": "La piscine à Mons en Baroeul est une piscine couve
        de larges baies vitrées et d'une mezzanine avec vue plongeante
        proposés au public.",
    "tel": "03 20 56 13 11",
    "adresse": "20 rue Lacordaire, 59370  Mons-en-Baroeul",
    "nom": "Piscine de Mons en Baroeul",
    "__v": 0
  }
]
```

DOCUMENT

DOCUMENT

TERMINOLOGIES SQL ET MONGODB

In SQL	In Mongo
• Database	• Database
• Table	• Collection
• Record	• Document

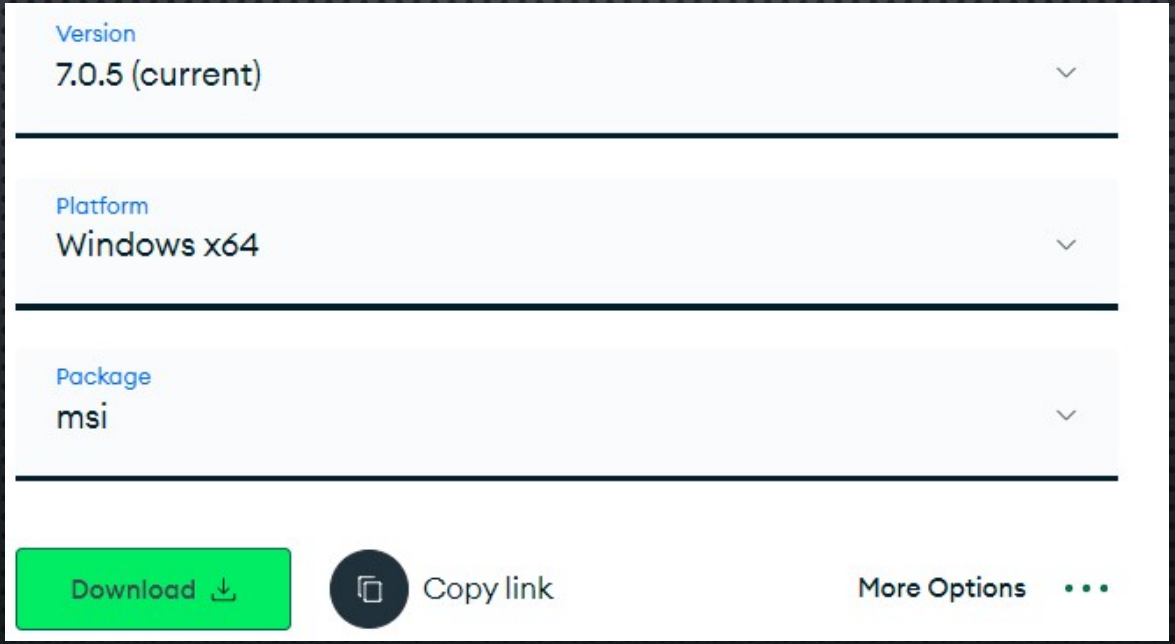
LES SOLUTIONS MONGODB

- MongoDB Atlas est la solution cloud de Database as a Service (DBaaS). Atlas vous déploie un serveur MongoDB managé sur un cloud.
- Installer en local : MongoDB peut être installé localement, ce qui vous permettra d'héberger votre propre serveur MongoDB sur votre matériel.

INSTALLER MONGODB

- Site :

https://www.mongodb.com/try/download/community



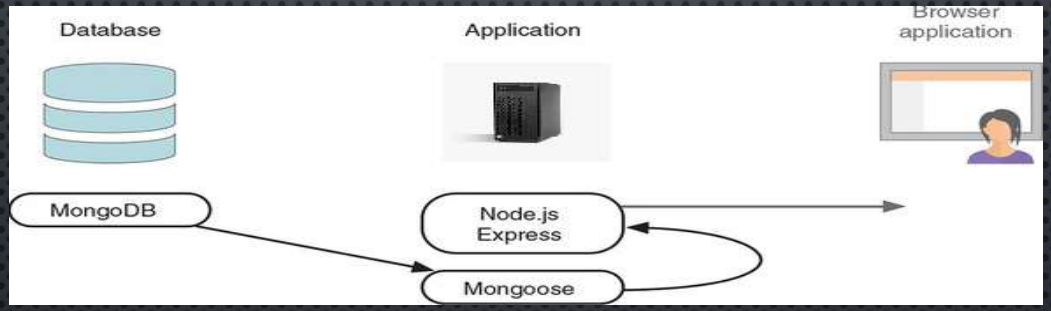
COMPASS

- MongoDB Compass est une interface graphique puissante pour interroger, agréger et analyser vos données MongoDB dans un environnement visuel.
- Compass est gratuit à utiliser et source disponible, et peut être exécuté sur macOS, Windows et Linux.
- Télécharger et installer :

https://www.mongodb.com/try/download/compass

MONGOOSE

- Afin de faciliter l'accès et les différentes opérations liées à la base MongoDB, nous allons utiliser Mongoose.
- Mongoose est ce qui va permettre à Node.js de communiquer avec le serveur MongoDB : L'application Node / Express interagit avec MongoDB via Mongoose.



- Mongoose c'est un ODM (Object Document Mapping), un object document model. C'est-à-dire que, concrètement, c'est une librairie qui va vous permettre de travailler avec des objets sous format objet avec des schémas d'objets, avec de la validation, directement via une librairie.

- Pour l'installer il suffit d'entrer la commande (dans votre console dans le projet Express)

npm install mongoose

EXEMPLE AVEC EXPRESSS ET MONGOOSE

- On commence par créer un dossier
- Dans le terminal taper :

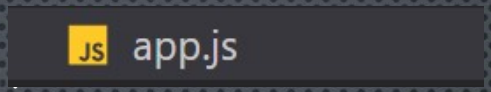
npm init –y

- Puis installer les dépendances

npm i express mongoose

CONNEXION À LA BASE DE DONNÉE ET AU SERVEUR

- Dans la racine du projet, on va créer le fichier app.js .




```
const express = require('express')
const mongoose = require('mongoose')
const app = express()

// Connexion à la base données
mongoose.connect("mongodb://127.0.0.1:27017/myDataBase")
    .then(() => {console.log("DataBase Successfully Connected");
   }).catch(err => {
    console.log('Unable to connect to database', err);
    process.exit();
   });
     app.get("/",(req,res)=>{
   res.send("bonjour");
   });
 // Connexion au serveur
   app.listen(3001, () => {
    console.log("Server is listening on port 3001"); });
```

DÉMARRER LES SERVEURS

- Exécutez l'application avec la commande suivante :

node app.js

Ou puisque nodemon est installé exécuter :

nodemon app

```
[nodemon] starting `node app.js`
Server is listening on port 3001
DataBase Successfully Connected
```

SCHEMA

- Tout dans Mongoose commence par un schéma. Chaque schéma correspond à une collection MongoDB et définit la forme des documents au sein de cette collection.
- Mongoose utilise des « Schemas » pour modéliser les données.
- Il permet de définir les types de variables et de structurer les données.
- Pour créer un Schema avec Mongoose, il suffit d'utiliser l'objet **Schema** :
- Exemple :

```
var articleSchema = new mongoose.Schema({
  auteur : Schema.ObjectId,
  contenu : String,
  date : { type : Date, default : Date.now },
});
```

MODEL

- La méthode Schema de Mongoose vous permet de créer un schéma de données pour votre base de données MongoDB.
- Pour utiliser notre définition de schéma, nous devons convertir notre blogSchema en un Model avec lequel nous pouvons travailler.
- La méthode model transforme ce modèle en un modèle utilisable.
- Un model permet d'insérer des données dans MongoDB en respectant le schéma précisé et d'utiliser des requêtes dessus.
- Pour créer un Model, on le passe dans :

mongoose.model(modelName, schema)

- Exemple :

```
var CommentaireArticleModel = mongoose.model('commentaires', commentaireArticleSchema);
```

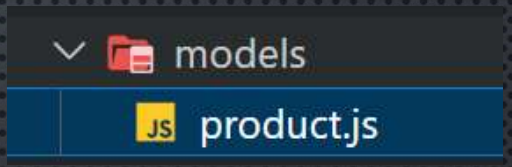
Cela va créer un model nommé « commentaires » à partir du schéma CommentaireArticleSchema et qui va le renvoyer dans la variable CommentaireArticleModel.

Le model va créer une collection automatiquement nommée « commentaires » (si elle n'existe pas dès qu'un élément sera inséré dedans).

- On peut également, créer le model dans un fichier .js en incluant Mongoose et en le récupérant dans un autre fichier

EXEMPLE MODEL

- On va créer un model appelé Product dans models/product.js :




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

const ProductSchema = new mongoose.Schema({
    name:String,
    price: Number,
})

const Product = mongoose.model('Product', ProductSchema)

module.exports = Product
```

LES INSTANCES DE MODEL

- Pour créer une instance de Model, il suffit de faire :

```
var monCommentaire = new CommentaireArticleModel({ pseudo : 'Alio' });
// On rajoute le contenu du commentaire, possible de le faire lors de l'instanciation
monCommentaire.contenu = 'Salut, super article sur Mongoose !';
```

Fonction save()

- La fonction save() de Mongoose sert à conserver un document dans la base de données MongoDB.
- Si la base de données n'existe pas, elle sera créée,
- En appelant cette fonction, vous pouvez ajouter de nouveaux documents à la base de données ou mettre à jour ceux existants.
- Exemple :

```
var new_user = new User({
   name: 'Mohamed',
   age:34
})

new_user.save()
```

QUERIES

Mongoose models fournit plusieurs fonctions d'assistance static pour CRUD operations . Chacune de ces fonctions renvoie un mongoose Query object .

- Model.deleteMany()
- Model.deleteOne()
- Model.find()
- Model.findById()
- Model.findByIdAndDelete()
- Model.findByIdAndRemove()
- Model.findByIdAndUpdate()
- Model.findOne()
- Model.findOneAndDelete()
- Model.findOneAndRemove()
- Model.findOneAndReplace()
- Model.findOneAndUpdate()
- Model.replaceOne()
- Model.updateMany()
- Model.updateOne()

Fonction find()

- La fonction find() est utilisée pour rechercher des données particulières dans la base de données MongoDB.
- Il faut 3 arguments : une requête (également appelée condition), une projection de requête (utilisée pour mentionner les champs à inclure ou à exclure de la requête), et le dernier argument correspond aux options générales de requête (comme limiter, ignorer, etc.) .
- **Exemple 1** :

```
const query = MyModel.find();
query.where('age').gte(21).exec();
```

- **Exemple 2** :

```
User.find({ age: {$gte:10}}, null, {limit:2}, function (err, docs) {
   if (err){
      console.log(err);
   }
   else{
      console.log("Third function call : ", docs);
   }
})
```

Fonction findById() et findOne()

- La fonction findById() est utilisée pour rechercher un seul document par son champ _id. Le champ _id est converti en fonction du schéma avant d'envoyer la commande.
- Exemple :

```
var id = '5ebadc45a99bde77b2efb20e';
User.findById(id)
```

- La fonction findOne() est utilisée pour rechercher un document selon la condition. Si plusieurs documents correspondent à la condition, il renvoie le premier document satisfaisant la condition.
- Exemple 1 :

```
User.findOne({age: {$gte:5} })
```

- Exemple 2 :

```
const query = Kitten.where({ color: 'white' });
const kitten = await query.findOne();
```

Fonction findByIdAndUpdate()

- La fonction findByIdAndUpdate() est utilisée pour rechercher un document correspondant, le met à jour en fonction de l'argument de mise à jour, en transmettant toutes les options, et renvoie le document trouvé.
- Exemple :

```
var user_id = '5eb985d440bd2155e4d788e2';

User.findByIdAndUpdate(user_id, { name: ‘Mohamed' })
```

Fonction findByIdAndDelete()

- La fonction findByIdAndDelete() est utilisée pour rechercher un document correspondant, le supprime et transmet le document trouvé.
- Exemple :

```
var id = '5ebadc45a99bde77b2efb20e';
User.findByIdAndDelete(id)
```

TRAITEMENT DE LA REQUÊTE

- Une requête mongoose peut être exécutée de deux manières. Premièrement, si vous transmettez une fonction callback , le Mongoose exécutera la requête de manière asynchrone et transmettra les résultats au callback .
- Une requête possède également une fonction .then() et peut donc être utilisée comme une promesse.
- Les requêtes Mongoose ne sont pas des promesses. Les requêtes sont thenables , ce qui signifie qu'elles ont une méthode .then() pour async/await pour plus de commodité.

EXEMPLE ROUTES ET QUERIES

- On va créer les routes sous routes/product.routes.js :

```
const express = require('express');
const router = express.Router();
const Product=require("../models/product");
```

```
// afficher la liste des products.
router.get('/', async (req, res, )=> {
     try {
    const products = await Product.find()
    res.status(200).json(products);
     } catch (error) {
    res.status(404).json({ message:
error.message });
    }
    });

// chercher un product
router.get('/:id',async(req, res)=>{
     try {
     const product = await
Product.findById(req.params.id);
    res.status(200).json(product);
    } catch (error) {
     res.status(404).json({ message:
error.message });
     }
    });
```

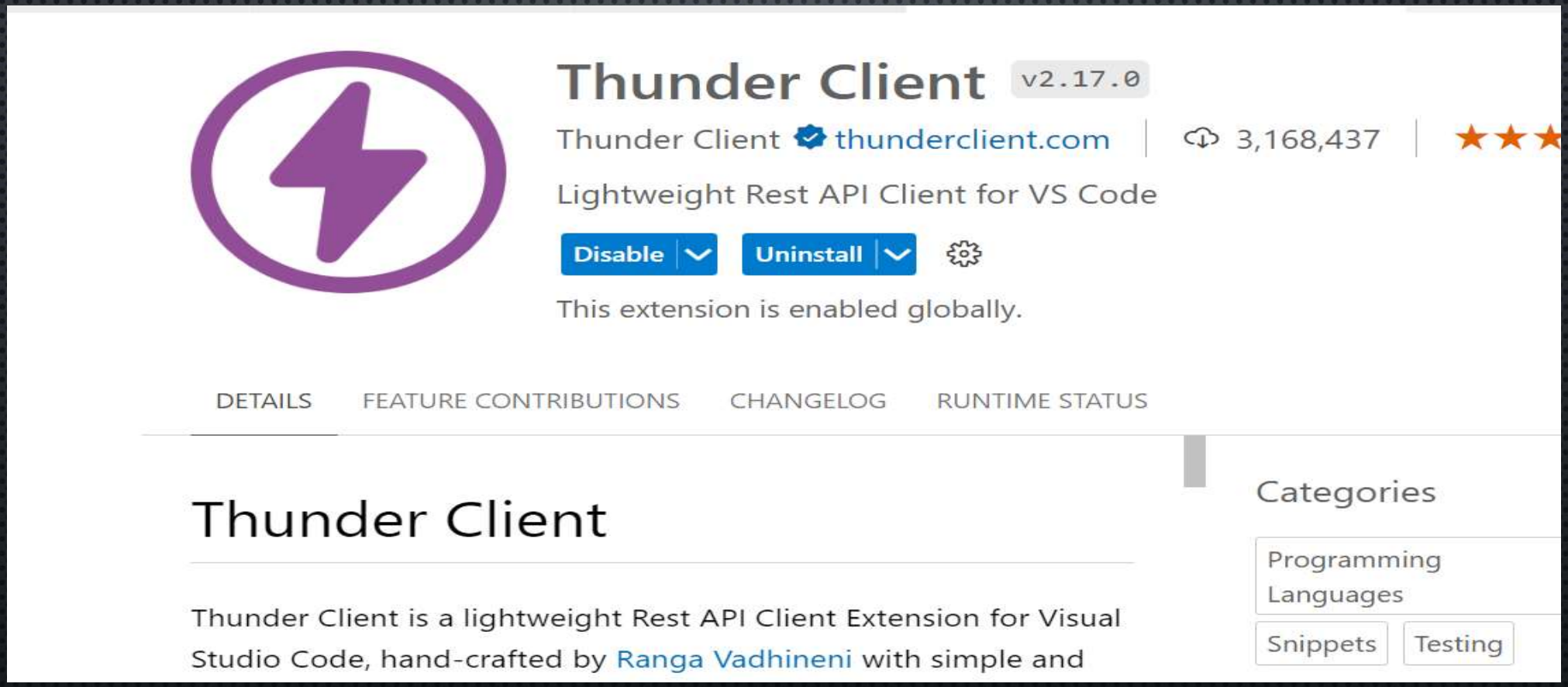
```
// modifier un product
router.put('/:id', async (req, res)=> {
    try {
    const {name,price}=req.body;
    const id = req.params.id;
    const newProd = {_id:id
,name:name,price:price}
    await Product.findByIdAndUpdate(id,
newProd);
    res.json(newProd);
    } catch (error) {
     res.status(404).json({ message:
error.message });
    }
    });

// Supprimer un product
router.delete('/:id', async (req, res)=> {
    await
Product.findByIdAndDelete(req.params.id);
    res.json({ message: " Product deleted
successfully." });
   });

module.exports = router;
```

APPEL DE ROUTES

- Dans **app.js** on va faire appel aux routes créées :

```
const express = require('express')
const mongoose = require('mongoose')
const app = express()

// Connexion à la base données
mongoose.connect("mongodb://127.0.0.1:27017/myDataBase")
    .then(() => {console.log("DataBase Successfully Connected");})
    .catch(err => { console.log("Unable to connect to database", err); process.exit(); });

app.get("/",(req,res)=>{
   res.send("bonjour");
   });

//BodyParser Middleware
app.use(express.json());

//Appel de routes
const productRoutes= require('./routes/product.routes');
app.use('/api/products',productRoutes);

// Connexion au serveur
   app.listen(3001, () => {
    console.log("Server is listening on port 3001"); });
```

TEST AVEC VSCODE THUNDER CLIENT



TEST POST

POST http://localhost:3001/api/products Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "name":"Notebook Samsung",
3   "price":800
4 }
```

Status: 200 OK Size: 80 Bytes Time: 36 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "name": "Notebook Samsung",
3   "price": 800,
4   "_id": "65b75e80a314d3429bf2c65a",
5   "__v": 0
6 }
```

TEST GET

GET http://localhost:3001/api/products Send

Status: 200 OK Size: 171 Bytes Time: 14 ms

Query Headers 2 Auth Body 1 Tests Pre Run

Response Headers 6 Cookies Results Docs

Query Parameters

parameter	value


```
[
  {
    "_id": "65b75e80a314d3429bf2c65a",
    "name": "Notebook Samsung",
    "price": 800,
    "__v": 0
  },
  {
    "_id": "65b75ff9a314d3429bf2c65c",
    "name": "Montre connectée HUAWEI",
    "price": 900,
    "__v": 0
  }
]
```

TEST GET ONE BY ID

GET http://localhost:3001/api/products/65b75e80a314d3429bf2 Send

Status: 200 OK Size: 80 Bytes Time: 9 ms

Query Headers 2 Auth Body 1 Tests Pre Run

Response Headers 6 Cookies Results Docs

Query Parameters

parameter	value


```
1 {
2   "_id": "65b75e80a314d3429bf2c65a",
3   "name": "Notebook Samsung",
4   "price": 800,
5   "__v": 0
6 }
```

TEST PUT

PUT http://localhost:3001/api/products/65b75e80a314d3429bf2 Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "name": "Notebook Samsung 10",
3   "price": 850
4 }
```

Status: 200 OK Size: 75 Bytes Time: 5 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "_id": "65b75e80a314d3429bf2c65a",
3   "name": "Notebook Samsung 10",
4   "price": 850
5 }
```

TEST DELETE

DELETE http://localhost:3001/api/products/65b75ff9a314d342 Send

Query Headers 2 Auth Body 1 Tests Pre Run

Query Parameters

parameter	value

Status: 200 OK Size: 44 Bytes Time: 12 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "message": " Product deleted successfully."
3 }
```

POPULATION : RÉFÉRENCES À D'AUTRES DOCUMENTS

- Il n'y a pas de jointure dans MongoDB mais parfois nous voulons quand même des références à des documents dans d'autres collections.
- C'est là qu'intervient population .
- Dans MongoDB, Population est le processus de remplacement du chemin spécifié dans le document d'une collection par le document réel de l'autre collection.
- Besoin de population : chaque fois que dans le schéma d'une collection, nous fournissons une référence (dans n'importe quel champ) à un document de n'importe quelle autre collection, nous avons besoin d'une méthode populate() pour remplir le champ avec ce document.

SCHEMA AVEC POPULATION

- Dans l'exemple suivant, nous avons un userSchema et un autre postSchema, dans postSchema nous avons un champpostedBy qui fait référence à un document du modèle User.

```
const userSchema = new mongoose.Schema({
    username: String,
    email: String
})
const postSchema = new mongoose.Schema({
    title: String,
    postedBy: {
        type: mongoose.Schema.Types.ObjectId,
        ref: "User"
    }
})
const User = mongoose.model('User', userSchema);
const Post = mongoose.model('Post', postSchema);
module.exports = {
    Source, Destination, User, Post
}
```

Find AVEC POPULATION

- Nous effectuerons la requête pour trouver toutes les publications en utilisant la méthode populate().
- Pour résoudre le problème ci-dessus, la méthode populate() est utilisée pour remplacer le champ ObjectId utilisateur par l'ensemble du document composé de toutes les données utilisateur.

```
Post.find()
.populate("postedBy")
```

```
[
  {
    _id: 60aa2602d9f6080550eccbda,
    title: 'first post',
    postedBy: {
      _id: 60aa2602d9f6080550eccbd9,
      username: 'Paul',
      email: 'paulD@gmail.com',
      __v: 0
    },
    __v: 0
  }
]
```

CRÉATION D'UNE API REST AVEC EXPRESS ET PRISMA

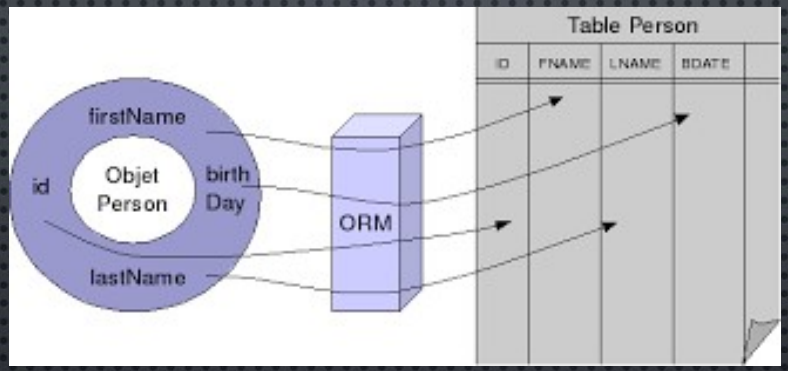
PRISMA

QU'EST-CE QUE PRISMA ?

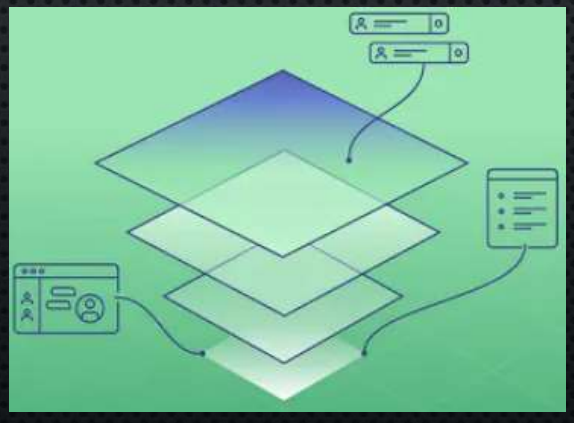
- Prisma est une couche d'accès aux données moderne et open source qui offre une manière flexible et intuitive de travailler avec des bases de données.
- Il vous permet d'interagir facilement avec vos données en fournissant une API propre et déclarative et des outils robustes pour créer des requêtes complexes.
- Prisma est un ORM /ODM de nouvelle génération qui facilite la garantie de la cohérence des données en fournissant un schéma facile à lire et un client de base de données de type sécurisé avec saisie semi-automatique pour toutes les requêtes.

ORM/ODM

- Prisma est un ORM (Object Relational Mapping) permettant d'avoir une correspondance entre un modèle objet et un modèle relationnel (SQL).



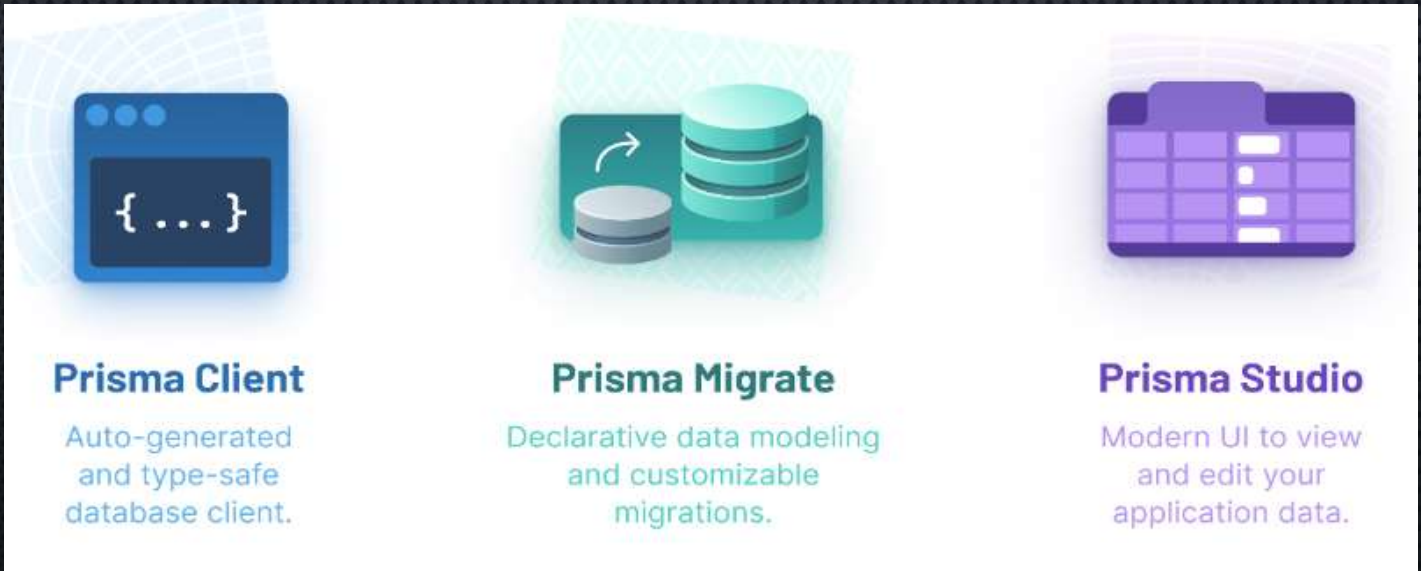
- Prisma est aussi un ODM (Object Document Mapping) qui permet aux développeurs de modéliser, d'interroger et de gérer plus facilement les données au-dessus des pilotes natifs de MongoDB. Il garantit la cohérence des données en permettant aux développeurs de formuler leur modèle de données dans des schémas faciles à lire.



COMPOSITION DE PRISMA

Prisma se compose des parties suivantes :

- **Client Prisma** : générateur de requêtes généré automatiquement et de type sécurisé pour Node.js et TypeScript
- **Prisma Migrate** : système de migration
- **Prisma Studio** : interface graphique pour afficher et modifier les données de votre base de données.



INSTALLATION PRISMA

- Commençons par installer l'interface en ligne de commande (CLI) de Prisma en tant que dépendance de développement :

npm install prisma

- La CLI étant installée, il est temps d'initialiser le projet Prisma :

npx prisma init

- Cette commande fait 2 choses :
  - Elle crée un dossier 📂 prisma contenant le fichier de configuration principal de l'ORM : 📄 schema.prisma.
  - Elle crée un fichier 📄 .env à la racine du projet contenant une variable DATABASE_URL dans laquelle nous irons spécifier les informations de connexion à la base de données.

CONFIGURER LA BASE DE DONNÉES

- Ouvrir le fichier **.env** puis changer la chaîne de connexion par défaut qui sera MySQL :

DATABASE_URL="mysql://root@localhost:3306/mybase"

PROVIDER DATASOURCE

- Dans le fichier **/prisma/schema.prisma** , pour connecter votre base de données, vous devez définir le champ url du bloc source de données dans votre schéma Prisma sur l'URL de connexion à votre base de données, une variable d'environnement qui a été définie dans .env
- Au niveau du provider est précisé le SGBD ici c'est mysql qu'il faut avoir,

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}
```

CRÉATION DU SCHÉMA DE BASE DE DONNÉES

- Il s'agit de définir les entités manipulées au sein de notre application, ainsi que leurs caractéristiques. Il peut s'agit de User, Article, Post, …
- **Modèles** : On définit un modèle de données avec le mot-clé **model** suivi du nom du modèle. Un nom de modèle Prisma doit correspondre au nom de la table en base de données.
- **Champs** : Les propriétés d'un modèle sont appelées champs. Un champ est défini par son nom, son type et des attributs (optionnels).
- **Type** d'un champ détermine la structure de données qu'il contient. On les écrit en PascalCase juste après le nom du champ.
- On distingue 2 grandes catégories de types de données :
  - Les types scalaires (scalar) : regroupant tous les types usuels (String, Int, Boolean…).
  - Les types de modèles (relations) : indiquant qu'un modèle est lié à un autre.
- Les attributs de champ sont préfixés du caractère **@** et permettent de modifier le comportement d'un champ spécifique.

TYPE

- Le type d'un champ détermine la structure de donnée qu'il contient. On les écrit en PascalCase juste après le nom du champ.

- On distingue 2 grandes catégories de types de données :

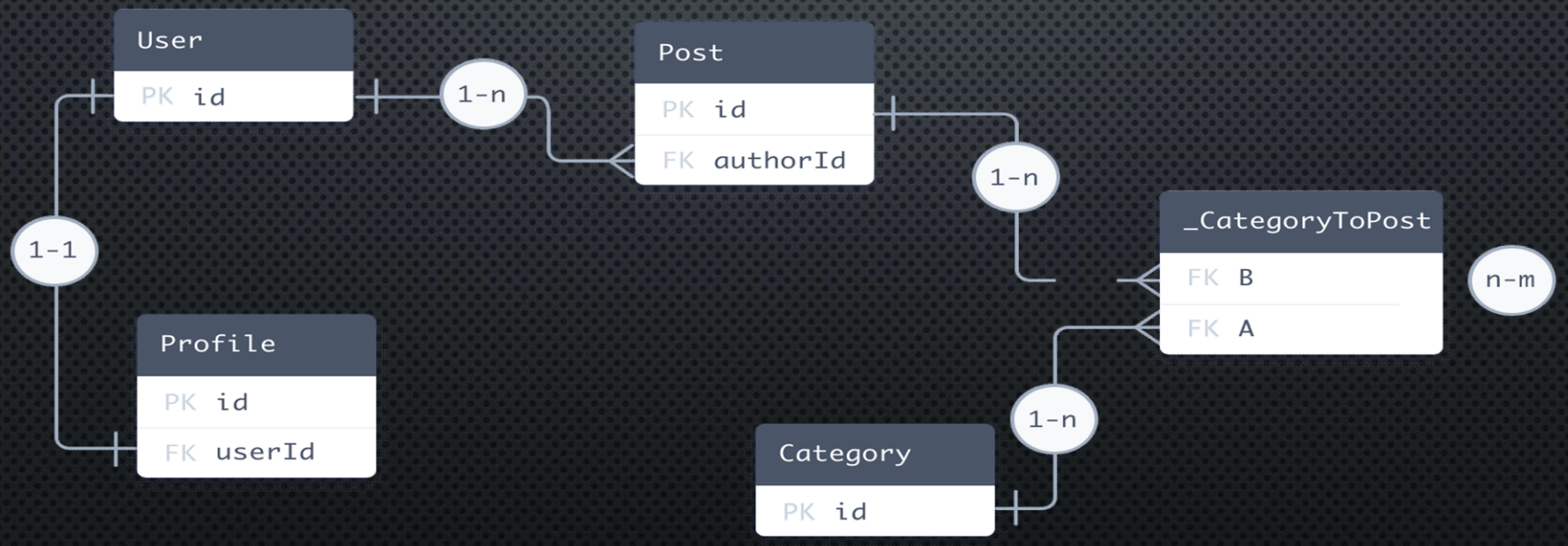
  - Les types scalaires (scalar) : regroupant tous les types usuels de la programmation et des bases de données (String, Int, Boolean…).
  - Les types de modèles (relations) : indiquant qu'un modèle est lié à un autre.

EXEMPLE DE SCHEMA

```
generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}
model Post {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  title     String   @db.VarChar(255)
  content   String?
  published Boolean  @default(false)
  author    User     @relation(fields: [authorId], references: [id])
  authorId  Int
}
model User {
  id      Int      @id @default(autoincrement())
  email   String   @unique
  name    String?
  posts   Post[]
}
```

RELATIONS

- Une relation consiste à connecter deux modèles Prisma.



RELATION OneToOne

Pour implémenter une relation OneToOne, il faut :

- Ajouter dans chaque modèle un champ relationnel, typé par le modèle lié.
- La relation étant 1-1, les noms de champs sont au singulier.
- Ajouter dans un des deux modèles un attribut scalaire **(Int @unique)** représentant la clé étrangère de la relation.
- On le nomme du même nom que le modèle qu'il référence, suivi de Id.
- Ajouter l'attribut **@relation** sur le champ relationnel du modèle possédant la clé étrangère.
- Ce champ possède un argument **fields** référençant la clé étrangère et un argument **references** référençant la clé primaire du modèle opposé.
- Le **?** indique qu'un utilisateur peut ne pas avoir de valeur renseignée.

EXEMPLE OneToOne

```
model User {
    id      Int      @id @default(autoincrement())
    // ...
    profile  Profile?
}

model Profile {
    id      Int   @id @default(autoincrement())
    // ...
    user    User  @relation(fields: [userId], references: [id])
    userId  Int   @unique
}
```

RELATIONS OneToMany / ManyToOne

Pour implémenter une relation OneToMany / ManyToOne, il faut :

- Ajouter dans chaque modèle un champ relationnel, typé par le modèle lié
- La relation étant 1-n, un nom de champ est au singulier et l'autre au pluriel (suivi du modificateur de type de liste **[ ]**).
- Ajouter dans le modèle se situant côté Many de la relation, un attribut scalaire (Int) représentant la clé étrangère de la relation.
- On le nomme du même nom que le modèle qu'il référence, suivi de Id.
- Ajouter l'attribut **@relation** sur le champ relationnel du modèle possédant la clé étrangère.
- Ce champ possède un argument **fields** référençant la clé étrangère et un argument **references** référençant la clé primaire du modèle opposé.

EXEMPLE OneToMany / ManyToOne

```
model User {
    id     Int     @id @default(autoincrement())
    // ...
    posts  Post[]
}

model Post {
    id        Int   @id @default(autoincrement())
    // ...
    author    User  @relation(fields: [authorId], references: [id])
    authorId  Int
}
```

RELATION ManyToMany

Au sein d'un schéma Prisma, on distingue 2 types de relations ManyToMany :

- Les relations implicites : constituées de 2 modèles ; aucun modèle n'est créé pour la table de jointure qui est alors implicite.
- Les relations explicites : constituées de 3 modèles ; on créé explicitement un modèle relationnel supplémentaire correspondant à la table de jointure. Cette approche est nécessaire si des champs additionnels sont à créer dans la table de jointure, comme par exemple une date associée à la relation.

RELATION ManyToMany IMPLICITE

- Pour implémenter une relation ManyToMany implicite, il faut simplement ajouter dans chaque modèle un champ relationnel, typé par l'entité liée.
- La relation étant n-n, les noms de champs sont au pluriel (suivi du modificateur de type de liste **[]**).

EXEPMLE ManyToMany IMPLICITE

```
model Post {
    id        Int       @id @default(autoincrement())
    // ...
    categories Category[]
}

model Category {
    id    Int    @id @default(autoincrement())
    // ...
    posts Post[]
}
```

RELATION ManyToMany EXPLICITE

Pour implémenter une relation ManyToMany explicite, il faut :

- Créer un modèle relationnel représentant la table de jointure. Ce modèle se nomme par convention : {Modèle A au pluriel}On{Modèle B au pluriel}.
- Ajouter au sein du modèle relationnel 2 champs relationnels typés par les modèles liés.
- Au sein d'une table de jointure, les relations vers les tables liées sont 1-1, les noms de champs sont donc au singulier.
- Ajouter dans le modèle relationnel, 2 attributs scalaire (Int @unique) pour les clés étrangères de la relation. On le nomme du même nom que le modèle qu'il référence, suivi de Id
- Définir ces modèles comme formant l'identifiant du modèle avec @@id
- Ajouter des champs au modèle relationnel si besoin.
- Ajouter l'attribut @relation sur les champ relationnels du modèle. Ces champs possèdent un argument fields référençant une clé étrangère et un argument references référençant la clé primaire du modèle opposé.
- Ajouter au sein des 2 modèles en relation, des attributs relationnels typés par le modèle relationnel.
- Les relations vers la table de jointure sont n-n, les noms de champs sont donc au pluriel (suivi du modificateur de type de liste []).

EXEMPLE ManyToMany EXPLICITE

```
model Post {
    id        Int              @id @default(autoincrement())
    // ...
    categories  CategoriesOnPosts[]
}

model Category {
    id    Int              @id @default(autoincrement())
    // ...
    posts  CategoriesOnPosts[]
}

model CategoriesOnPosts {
    post       Post     @relation(fields: [postId], references: [id])
    postId     Int
    category    Category  @relation(fields: [categoryId], references: [id])
    categoryId  Int
    assignedAt  DateTime  @default(now())
    @@id([postId, categoryId])
}
```

MIGRATION

- Il s'agit d'un processus de contrôle visant à synchroniser le schéma de données de l'application (défini ici dans 🗎 schema.prisma) avec la structure du schéma de base de données.
- Elles contribuent ainsi à la bonne évolution de votre modèle de données au fur et à mesure du développement de votre application.
- Ce changement d'état peut être caractérisé par : l'ajout d'une table, la modification d'une colonne, l'ajout d'une contrainte relationnelle, etc,
- Prisma Migrate vous met également en garde si une migration est potentiellement amenée à supprimer des données de la base. Ce qui est par exemple le cas si je décide de supprimer une colonne d'une table.
- Dans un environnement de développement, pour générer une migration Prisma, on exécute la commande :

npx prisma migrate dev

EXPLORER

OPEN EDITORS

BACKENDPRISMAMYSQL

- node_modules
- prisma
  - migrations
  - schema.prisma
- .env
- .gitignore
- package-lock.json
- package.json

schema.prisma X

prisma > schema.prisma

```prisma
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

model Post {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  title     String   @db.VarChar(255)
  content   String?
  published Boolean  @default(false)
  author    User     @relation(fields: [authorId], references: [id])
  authorId  Int
}

model User {
  id      Int      @id @default(autoincrement())
  email   String   @unique
  name    String?
  posts   Post[]
}
```

npx prisma migrate dev

```
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Datasource "db": MySQL database "mybase" at "localhost:3306"

MySQL database mybase created at localhost:3306

√ Enter a name for the new migration: ... migrateposts
Applying migration `20240115153433_migrateposts`

The following migration(s) have been created and applied from new schema changes:

migrations/
  └─ 20240115153433_migrateposts/
    └─ migration.sql

Your database is now in sync with your schema.

Running generate... (Use --skip-generate to skip the generators)

added 1 package, and audited 8 packages in 14s

found 0 vulnerabilities

✔ Generated Prisma Client (v5.8.0) to .\node_modules\@prisma\client in 153ms
```

EXPLORER

OPEN EDITORS

BACKENDPRISMAMYSQL

- node_modules
- prisma
  - migrations
    - 20240115153433_migrateposts
      - migration.sql
  - migration_lock.toml
  - schema.prisma
- .env
- .gitignore
- package-lock.json
- package.json

schema.prisma | migration.sql X

prisma > migrations > 20240115153433_migrateposts > migration.sql

```mysql
-- CreateTable
CREATE TABLE `Post` (
    `id` INTEGER NOT NULL AUTO_INCREMENT,
    `createdAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
    `updatedAt` DATETIME(3) NOT NULL,
    `title` VARCHAR(255) NOT NULL,
    `content` VARCHAR(191) NULL,
    `published` BOOLEAN NOT NULL DEFAULT false,
    `authorId` INTEGER NOT NULL,

    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

-- CreateTable
CREATE TABLE `User` (
    `id` INTEGER NOT NULL AUTO_INCREMENT,
    `email` VARCHAR(191) NOT NULL,
    `name` VARCHAR(191) NULL,

    UNIQUE INDEX `User_email_key`(`email`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

-- AddForeignKey
ALTER TABLE `Post` ADD CONSTRAINT `Post_authorId_fkey` FOREIGN KEY
```

phpMyAdmin

Récentes | Préférées

- Nouvelle base de données
- bibliothequedb
- information_schema
- mybase
  - Nouvelle table
  - post
  - user
  - _prisma_migrations

Serveur : 127.0.0.1 » Base de données : mybase » Table : post

Parcourir | Structure | SQL | Rechercher | Insérer | Exporter

MySQL a retourné un résultat vide (c'est à dire aucune ligne). (traitement en 0,0009 seconde(s).)

SELECT * FROM `post`

Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]

id	createdAt	updatedAt	title	content	published	authorId

Opérations sur les résultats de la requête

Créer une vue

PRISMA CLIENT

- Prisma Client génère le client utilisé dans le code de l'application.
- Selon l'équipe, l'un des principaux avantages de travailler avec Prisma Client est qu'il permet aux développeurs de penser en objets et offre donc une manière familière et naturelle de raisonner sur leurs données.
- Prisma Client n'a pas le concept d'instances de modèle.
- Au lieu de cela, il aide à formuler des requêtes de base de données qui renvoient toujours des objets JavaScript simples.
- Grâce aux types générés, vous bénéficiez également de l'autocomplétion pour ces requêtes.

INSTALLER LE CLIENT

- Installer Prisma Client dans votre projet avec la commande suivante :

```
npm install @prisma/client
```

Remarque : Si vous avez déjà exécuté le migrate avant d'exécuter la commande ci-dessus, le client Prisma est déjà installé car la génération de la migration l'exige, la CLI l'installe donc implicitement.

- Importer Prisma Client :

```
const { PrismaClient } = require('@prisma/client')

const prisma = new PrismaClient()
```

UTILISER LE CLIENT

- Les requêtes exécutées vers la base de données auront la forme :

const variable = await prisma.{model}.{method}

  - variable stocke le retour de la méthode
  - {model} correspond au modèle à interroger (en camelCase)
  - {method} correspond à la méthode du client Prisma à appeler

- Voilà un exemple concret, récupérant tous les utilisateurs de la BDD :

```
const users = await prisma.user.findMany({})
```

EXEMPLE CRUD

- Exploitons le client Prisma afin d'implémenter les opérations élémentaires d'un CRUD.

- Pour exploiter le client Prisma, créons un script Node.js élémentaire 📄 **app.js** à exécuter en ligne de commande.

  1. installer express

     npm i express

  2. créer 📄 app.js
  3. exécuter le serveur

     nodemon app

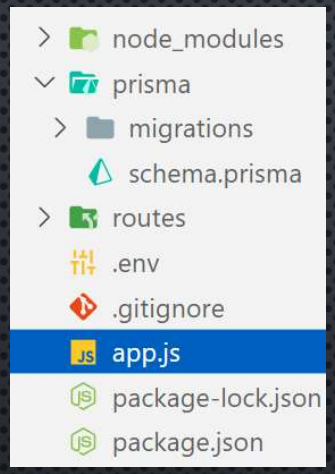
LA ROUTE post.route.js

Créer le dossier /routes ayant le fichier post.route.js

```
const express = require('express');
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const router = express.Router();

// afficher la liste des posts.
router.get('/', async (req, res, )=> {
    try {
        const posts = await prisma.Post.findMany()
            res.json(posts)
      } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
        })
      }
});
module.exports = router;
```

app.js

Créer le fichier app.js




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

const app = express()

app.use(express.json())

// Appel de routes
const postRouter =require("./routes/post.route")
app.use('/api/posts', postRouter);

const PORT = process.env.PORT || 3001
app.listen(PORT, () => console.log(`Server is running on port
${PORT}`))
```

EXÉCUTER LE SERVEUR NODEJS

http://localhost:3001/api/posts

localhost:3001/api/posts


```
[
    {
        "id": 1,
        "createdAt": "2024-01-15T17:05:05.000Z",
        "updatedAt": "2024-01-15T17:05:05.000Z",
        "title": "First post",
        "content": "this is the first post made",
        "published": false,
        "authorId": 1
    },
    {
        "id": 2,
        "createdAt": "2024-01-15T17:12:50.000Z",
        "updatedAt": "2024-01-15T17:12:50.000Z",
        "title": "second post",
        "content": "my second post made",
        "published": true,
        "authorId": 1
    }
]
```

findMany()

- findMany() retourne une liste de résultats.
- Cette méthode possède 8 options :
  - **where** (optionnel) : spécifie des critères de filtre.
  - **orderBy** (optionnel) : trie les enregistrement selon une propriété.
  - **skip** (optionnel) : ignore un nombre d'éléments retournés.
  - **cursor** (optionnel) : spécifie une position de départ à partir de laquelle récupérer les enregistrements.
  - **take** (optionnel) : spécifie combien d'objets récupérer depuis le début / la fin de la liste ou le curseur s'il est spécifié (avec cursor)
  - **select** (optionnel) : spécifie quels champs récupérer.
  - **include** (optionnel) : récupère les champs d'un modèle lié.
  - **distinct** (optionnel) : filtre les doublons pour une propriété spécifique.
- Si aucune option n'est spécifiée, alors findMany({}) retourne tous les enregistrements de la table.

EXEMPLE 1 findMany()

Créer le dossier /routes ayant le fichier user.route.js

```
const express = require('express');
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const router = express.Router();
router.get('/', async (req, res, )=> {
    try {   const users = await prisma.user.findMany({
            where: { email: {endsWith: "@gmail.com" }
            },
            select: { email: true },
            orderBy: { id: "desc" },
            take: 5
        })
    res.json(users)
     } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
        })
     }
});
module.exports = router;
```

Appel de la route dans app.js

```
const userRouter
=require("./routes/user.route")
app.use('/api/users', userRouter);
```

Résultat

localhost:3001/api/users


```
[
    {
        "email": "alia@gmail.com"
    },
    {
        "email": "farah@gmail.com"
    },
    {
        "email": "ali@gmail.com"
    },
    {
        "email": "mohamed@gmail.com"
    },
    {
        "email": "hassan@gmail.com"
    }
]
```

EXEMPLE 2 findMany() avec include

Dans routes/post.route.js

```
router.get('/', async (req, res, )=> {
    try {
      const posts = await prisma.post.findFirst({
        include: {
          author: true,
        },
      })

        res.json(posts)
      } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
        })
      }

});
```

Résultat

localhost:3001/api/posts


```
{
    "id": 1,
    "createdAt": "2024-01-15T17:05:05.000Z",
    "updatedAt": "2024-01-15T17:05:05.000Z",
    "title": "First post",
    "content": "this is the first post made",
    "published": false,
    "authorId": 1,
    "author": {
        "id": 1,
        "email": "hassan@gmail.com",
        "name": "Hassan"
    }
}
```

findUnique()

- findUnique() retourne un résultat unique en fonction d'un ou plusieurs critère(s) de sélection unique (tel qu'un identifiant ou une adresse email).

- Cette méthode possède 3 options :

  - **where** (obligatoire) : spécifie des critères de filtre.
  - **select** (optionnel) : spécifie quels champs récupérer.
  - **include** (optionnel) : récupère les champs d'un modèle lié.

EXEMPLE findUnique()

/routes/user.route.js

```
router.get('/:id', async (req, res, )=> {
    const { id } = req.params
     try {
        const users = await prisma.user.findUnique({
            where: {    id: Number(id)  },
            select: {   email: true }
        })
      res.json(users)
      } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
        })
      }

});
```

Résultat

localhost:3001/api/users/1

```
1 {
2     "email": "hassan@gmail.com"
3 }
```

create()

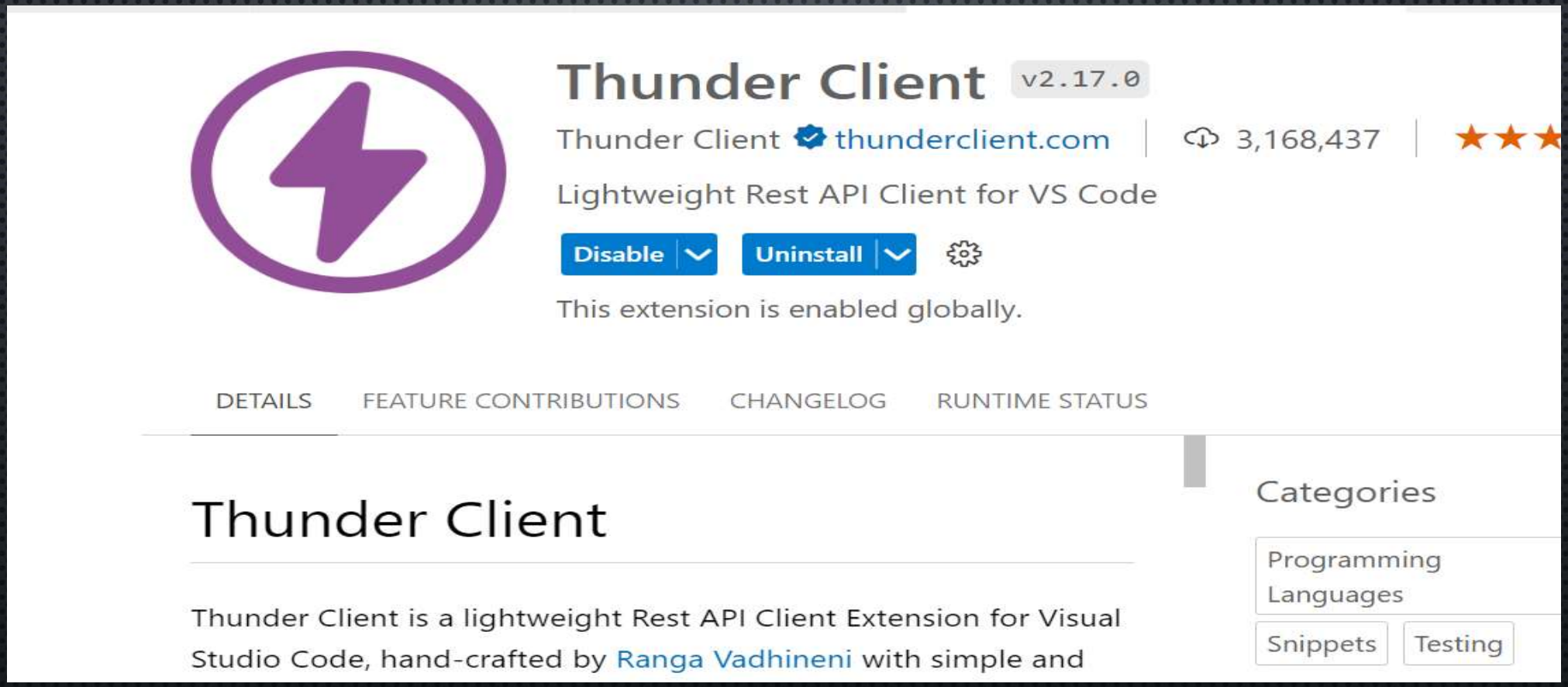
- Pour insérer des enregistrements en base de données, on exploite la méthode create()
- Cette méthode possède 3 options :
  - **data** (obligatoire) : l'objet à insérer en base de données. Les champs marqués comme facultatifs ou ayant des valeurs par défaut dans le modèle de données sont facultatifs.
  - **select** (optionnel) : spécifie les propriétés à inclure sur l'objet retourné.
  - **include** (optionnel) : spécifie les relations à construire sur l'objet retourné.
- Il est possible d'effectuer une insertion multiple en bouclant sur la méthode create() ou si les conditions le permettent, d'utiliser la méthode createMany().

EXEMPLE create()

/routes/user.route.js

```
router.post('/', async (req, res, )=> {
    const {name,email}=req.body
    try {
        const user = await prisma.user.create({
            data: {
                name: name,
                email: email
            }
        })
       res.json(user)
      } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
        })
      }
});
```

L'EXTENSION VSCODE THUNDER CLIENT



REQUÊTE POST

POST http://localhost:3001/api/users Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "name":"Samia",
3   "email":"samia@gmail.com"
4 }
```

Status: 200 OK Size: 49 Bytes Time: 141 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "id": 6,
3   "email": "samia@gmail.com",
4   "name": "Samia"
5 }
```

update()

- Pour modifier des enregistrement en base de données, on exploite la méthode update()
- Cette méthode possède 4 options :
  - **data** (obligatoire) : l'objet à insérer en base de données. Les champs marqués comme facultatifs ou ayant des valeurs par défaut dans le modèle de données sont facultatifs.
  - **where** (obligatoire) : spécifie des critères de filtre pour sélectionner l'objet à modifier.
  - **select** (optionnel) : spécifie les propriétés à inclure sur l'objet retourné.
  - **include** (optionnel) : spécifie les relations à construire sur l'objet retourné.
- Si vous souhaitez modifier un ensemble d'enregistrements, vous pouvez utiliser la méthode updateMany().

EXEMPLE update()

/routes/user.route.js

```
router.put('/:id', async (req, res)=> {
    const { email,name} = req.body;
    const id  = req.params.id;

    try {
     const user = await prisma.user.update({
            data: { email: email , name : name },
            where: { id: Number(id)},
        })
    res.json(user);
    } catch (error) {
    res.status(404).json({ message: error.message });
    }
});
```

REQUÊTE PUT

PUT http://localhost:3001/api/users/6 Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1  {
2    "name":"Samira",
3    "email":"samira@gmail.com"
4  }
```

Status: 200 OK Size: 51 Bytes Time: 49 ms

Response Headers 6 Cookies Results Docs

```
1  {
2    "id": 6,
3    "email": "samira@gmail.com",
4    "name": "Samira"
5  }
```

delete()

- Pour supprimer des enregistrements en base de données, on exploite la méthode delete()
- Cette méthode possède 3 options :
  - **where** (obligatoire) : des critères de filtre pour sélectionner l'objet à supprimer.
  - **select** (optionnel) : spécifie les propriétés à inclure sur l'objet retourné.
  - **include** (optionnel) : spécifie les relations à construire sur l'objet retourné.
- Si vous souhaitez supprimer un ensemble d'enregistrements, vous pouvez utiliser la méthode deleteMany().

EXEMPLE delete()

/routes/user.route.js

```
router.delete('/:id', async (req, res)=> {
    const  id  = req.params.id;
    try {
     await prisma.user.delete({
        where: { id: Number(id) },
    })

    res.json({ message: "user "+ id +" deleted successfully."
});
} catch (error) {
    res.status(404).json({ message: error.message });
    }
});
```

REQUÊTE DELETE

DELETE | http://localhost:3001/api/users/6 | Send

Status: 200 OK Size: 42 Bytes Time: 49 ms

Query Headers 2 Auth Body 1 Tests Pre Run

Response Headers 6 Cookies Results Docs

Query Parameters

parameter	value

```
1  {
2    "message": "user 6 deleted successfully."
3  }
```

FILTRAGE

- Nombreuses sont les requêtes qui nécessitent de filtrer les données.

- Lecture, modification ou encore suppression d'un enregistrement…
- Si filtrer un élément par son identifiant est un besoin récurrent, il est aussi parfois nécessaire de filtrer un ou plusieurs éléments sur n'importe laquelle de ses propriétés.

- Ce filtrage est rendu possible avec le mot-clé **where**.

OPÉRATEURS DE COMPARAISON PRINCIPAUX

Opérateur	Description
`equals`	Vérifie une égalité `=`
`not`	Vérifie une différence `≠`
`lt`	Vérifie une infériorité stricte `<`
`lte`	Vérifie une infériorité ou égalité `≤`
`gt`	Vérifie une supériorité stricte `>`
`gte`	Vérifie une supériorité ou égalité `≥`

OPÉRATEURS DE COMPARAISON SECONDAIRES

Opérateur	Description
`in`	Vérifie si une valeur est dans une liste
`notIn`	Vérifie si une valeur n'est pas dans une liste
`contains`	Vérifie la présence de caractères dans un ensemble
`search`	Vérifie la présence de caractères dans un ensemble (accepte les opérateurs logiques : `&`, `|` et `!`). On parle de full-text-search
`startsWith`	Vérifie la présence d'une chaîne de caractères au début d'une autre
`endsWith`	Vérifie la présence d'une chaîne de caractères à la fin d'une autre

OPÉRATEURS LOGIQUES

Opérateur	Description
AND	Il implique que toutes les conditions doivent être vérifiées
OR	Il implique qu'au moins une condition doit être vérifiée
NOT	Il implique que toutes les conditions ne doivent pas être vérifiées

FILTRES DE RELATION

Opérateur	Description
`some`	Retourne tous les enregistrements pour lesquels au moins 1 enregistrement lié vérifie un critère.
`every`	Retourne tous les enregistrements pour lesquels tous les enregistrements liés vérifient un critère.
`none`	Retourne tous les enregistrements pour lesquels aucun enregistrement lié ne vérifie un critère.

TRI

- Il est très fréquent de devoir trier un résultat. Le tri est défini à travers l'option Prisma orderBy. Un tri peut être ascendant ou descendant :

- ascendant : le mot-clé **asc** permet de définir un tri ascendant.
- descendant : le mot-clé **desc** permet de définir un tri descendant.

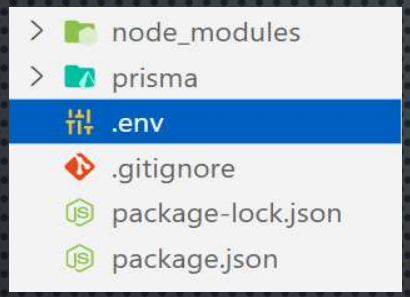
PRISMA AVEC MONGODB

CRÉATION DU PROJET

Dans le terminal exécuter :

- npm init -y
- npm install prisma
- npx prisma init
- npm i express

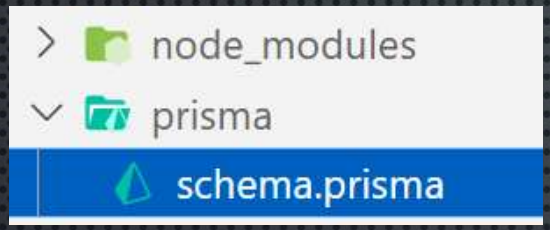
.env



DATABASE_URL="mongodb+srv://hassan:messk2015@cluster0.yfhhu.mongodb.net/DatabaseCommerce25?retryWrites=true&w=majority"

PORT=3001

SCHEMA




```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}
```

```
model articles {
  id            String       @id @default(auto()) @map("_id") @db.ObjectId
  designation  String       @unique(map: "designation_1")
  imageart     String
  marque       String
  prix         Float
  qtestock     Int
  reference    String       @unique(map: "reference_1")
  scategorieID String       @db.ObjectId
  scategories  scategories  @relation(fields: [scategorieID], references: [id])
  createdAt    DateTime?    @db.Date
  updatedAt    DateTime     @db.Date
}

model categories {
  id              String       @id @default(auto()) @map("_id") @db.ObjectId
  imagecategorie String
  nomcategorie   String       @unique(map: "nomcategorie_1")
  createdAt      DateTime?    @db.Date
  updatedAt      DateTime?    @db.Date
  scategories    scategories[]
}

model scategories {
  id               String      @id @default(auto()) @map("_id") @db.ObjectId
  imagescategorie String
  nomscategorie   String
  categorieID     String
  categories       categories  @relation(fields: [categorieID], references: [id])
  createdAt       DateTime?  @db.Date
  updatedAt       DateTime?  @db.Date
  articles        articles[]
}
```

REMARQUES

- Avec MongoDB nous n'avons pas besoin de faire migrate
- La base et les documents seront créés lors de la première requête POSt effectuée
- Le code utilisé dans les routes avec le cas de Mysql est valable avec MongoDB
- Définition des ID : les documents MongoDB ont un champ _id (qui contient souvent un ObjectID). Prisma ne prend pas en charge les champs commençant par _, ceux-ci doivent donc être mappés à un champ Prisma à l'aide de l'attribut @map.
- Cette structure de données représente une relation un-à-plusieurs.
- Dans les queries lorsqu'on a un test where id il ne faut plus mettre Number comme dans le cas de MySQL.

PRISMA CLIENT

- Dans le terminal exécuter :

npm i @prisma/client

- Prisma Client JS est un générateur de requêtes généré automatiquement qui permet un accès à la base de données de type sécurisé et réduit le passe-partout.
- Il fait partie de l'écosystème Prisma. Prisma fournit des outils de base de données pour l'accès aux données, la modélisation déclarative des données, les migrations de schémas et la gestion visuelle des données.

routes/categories.route.js

```
const express = require('express');
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const router = express.Router();
//Ajout d'une categorie
router.post('/', async (req, res, )=> {
    const {nomcategorie,imagecategorie}=req.body
    try {
        const categorie = await prisma.categories.create({
            data: {
                nomcategorie: nomcategorie,
                imagecategorie: imagecategorie
            }
        })
            res.json(categorie)
      } catch (error) {
        res.status(500).json({ message: error.message });
      }
});
```

```
// afficher la liste des catégories.
router.get('/', async (req, res, )=> {
    try {
        // categories est le nom du model précisé dans prisma.schema
        const categories = await prisma.categories.findMany()
         res.json(categories)
        } catch (error) {
         res.status(500).json({ message: error.message });
       }
});
```

app.js

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

const app = express()

//BodyParser Middleware
app.use(express.json());

// Appel de routes
const categoriesRouter =require("./routes/categories.route")
app.use('/api/categories', categoriesRouter);

const PORT = process.env.PORT || 3001

app.listen(PORT, () => console.log(`Server is running on port ${PORT}`))
```

TEST API POST

POST http://localhost:3001/api/categories/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1  {
2    "nomcategorie": "Électroménager",
3    "imagecategorie": "http://res.cloudinary.com/iset
       -sfax/image/upload/v1658749312/images/FerEvertek
       .jpg.jpg"
4  }
```

Status: 200 OK Size: 173 Bytes Time: 2.47 s

Response Headers 6 Cookies Results Docs { } Copy

```
1  {
2    "id": "65b519c2de5d31a36efba1ca",
3    "imagecategorie": "http://res.cloudinary.com/iset-sfax
       /image/upload/v1658749312/images/FerEvertek.jpg.jpg"
       ,
4    "nomcategorie": "Électroménager"
5  }
```

TEST API GET

GET http://localhost:3001/api/categories/ Send

Status: 200 OK Size: 175 Bytes Time: 2.60 s

Query Headers 2 Auth Body 1 Tests Pre Run

Response Headers 6 Cookies Results Docs { }

Query Parameters

parameter value

Copy


```
1  [
2    {
3      "id": "65b519c2de5d31a36efba1ca",
4      "imagecategorie": "http://res.cloudinary.com/iset-sfax
         /image/upload/v1658749312/images/FerEvertek.jpg
         .jpg",
5      "nomcategorie": "Électroménager"
6    }
7  ]
```