

NodeJS & MongoDB

Hedi Rivas - 11/2023

Programme

- Historique
- Installation
- npm et package.json
- Quelques commandes de base
- MongoDB : Atlas & Compass
- Mongoose
- Schéma & Modèle
- Modélisation & Relations
- Démarrer avec Express
- Les services tiers
- MVC & Modules



Historique



Historique

Node.js a été créé par Ryan Dahl en 2009. L'objectif était de créer quelque chose de plus léger et plus efficace que les environnements de serveur traditionnels comme Apache HTTP Server.

Il permet aux développeurs d'utiliser JavaScript, un langage traditionnellement client-side, pour écrire des logiciels côté serveur.

Les avantages de NodeJS

- NodeJS permet d'exécuter du code JavaScript sans passer par un navigateur.
- Extension des fonctionnalités de base grâce aux modules.
- Une suite d'outil en ligne de commande pour interagir avec le projet.

Installation



Installation

Pour pouvoir utiliser nodeJS sur sa machine, il est d'abord nécessaire de l'installer : voici le lien pour le télécharger <https://nodejs.org/en>.

⚠️ Attention veillez à télécharger la version LTS qui est la dernière version la plus stable.

npm et package.json



npm et package.json

Un élément clé de l'écosystème Node.js est npm, un gestionnaire de paquets qui permet aux développeurs d'installer et de gérer des bibliothèques et des outils tiers facilement.

Le `package.json` quant à lui est la carte d'identité de votre projet.

```
{
  "name": "myapp",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "mongoose": "^6.2.11"
  }
}
```

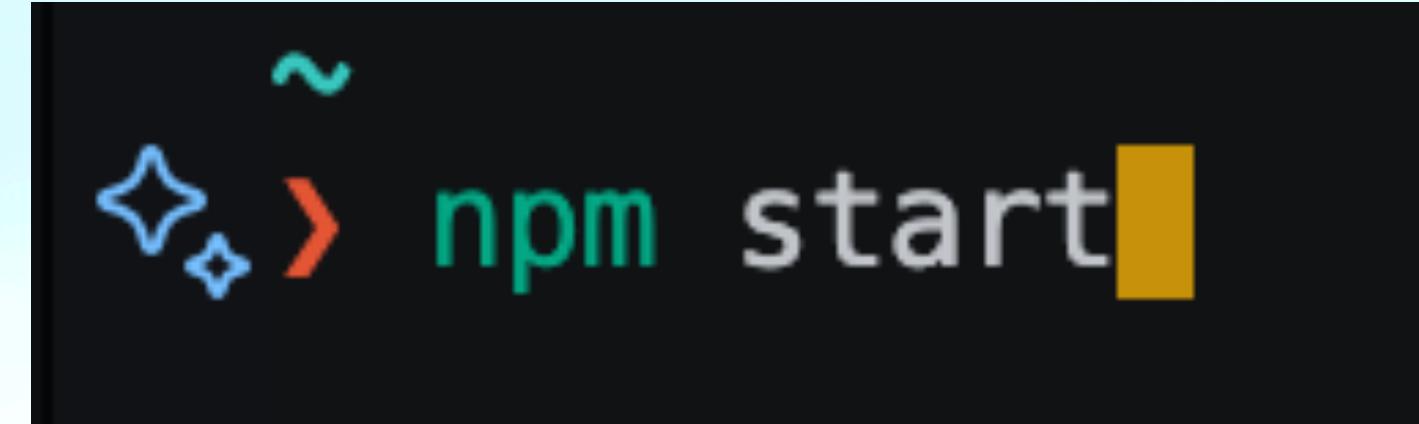
Quelques commandes de base



Quelques commandes de base

Démarrer un projet NodeJS

Une fois le projet initialisé, vous pouvez le lancer avec la commande npm start. Cette commande va exécuter l'ensemble des fichiers présents dans votre projet.



```
~ > npm start
```

package.json

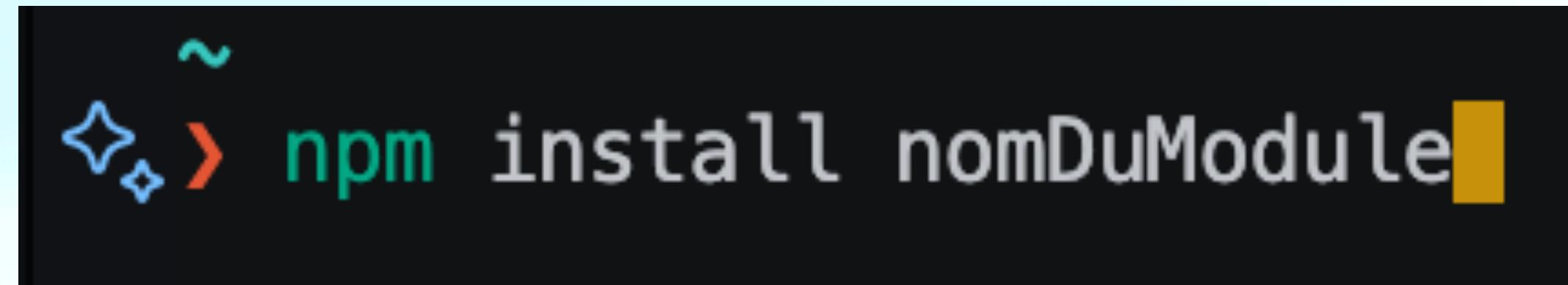
```
{  
  "name": "myapp",  
  "private": true,  
  "scripts": {  
    "start": "node app.js"  
  },  
  "dependencies": {  
    "mongoose": "^6.2.11"  
  }  
}
```

A red arrow points from the terminal window to the "start" script definition in the package.json file.

Quelques commandes de base

Ajout d'un module dans votre projet

L'ajout de module dans le projet se fait avec la commande npm install + le nom du module. Après installation, le nom du module apparaît dans le fichier package.json.



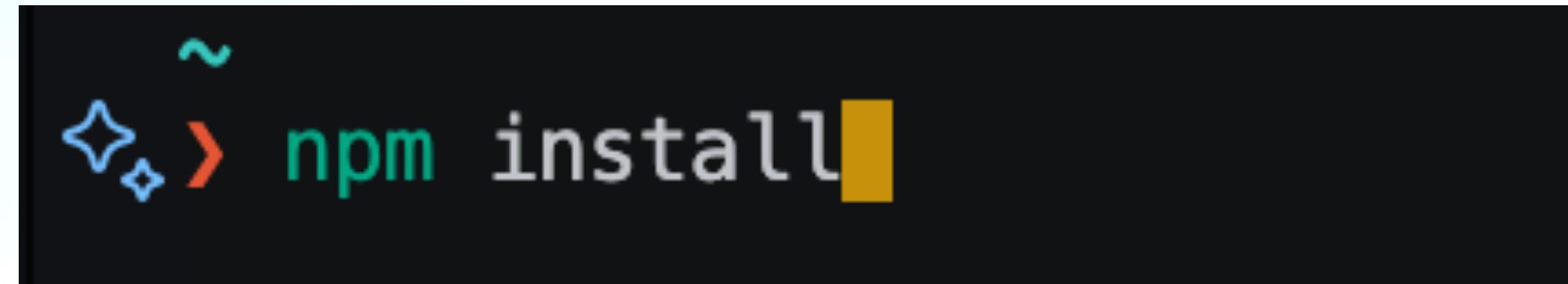
package.json

```
{  
  "name": "myapp",  
  "private": true,  
  "scripts": {  
    "start": "node app.js"  
  },  
  "dependencies": {  
    "mongoose": "^6.2.11"  
  }  
}
```

Quelques commandes de base

Installer les dépendances d'un projet

Enfin, pour installer les dépendances d'un projet il faut utiliser la commande npm install. Cette commande lit le package.json et installe toutes les dépendances manquantes.



MongoDB : Atlas & Compass



MongoDB®

MongoDB : Atlas & Compass

Contexte

Jusqu'ici les données créées sont effacées et réinitialisées à chaque exécution du code...

Nous allons donc utiliser une base de données pour rendre les données durables. Nous allons utiliser la base de données MongoDB.

MongoDB est une base de données dite noSQL, elle permet de gérer des données qui ne sont pas structurées (qui n'ont pas de relations les unes avec les autres).

MongoDB : Atlas & Compass

Fonctionnement

Les informations sont enregistrées au format JSON.

```
{ firstName: "John" , lastName: "Doe", age: 25 }

{ firstName: "Vanessa", lastName: "Smith", age: 38 }

{ firstName: "Mike", lastName: "Awesome", age: 42 }
```

MongoDB : Atlas & Compass

Vocabulaire de MongoDB

Un seul objet est appelé un **document**.

```
{ firstName: "John" , lastName: "Doe", age: 25 }
```

Un ensemble de document de la même famille est appelé une **collection**.

```
{ firstName: "John" , lastName: "Doe", age: 25 }
{ firstName: "Vanessa", lastname: "Smith", age: 38 }
{ firstName: "Mike", lastname: "Awesome", age: 42 }
```

MongoDB : Atlas & Compass

MongoDB Atlas

C'est un service gratuit et clé en main permettant de créer des bases de données.

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



MongoDB : Atlas & Compass

MongoDB Compass

C'est un logiciel permettant d'interagir facilement avec une BDD afin de l'administrer et modifier directement les données.

Lien : <https://www.mongodb.com/products/tools/compass>



MongoDB : Atlas & Compass

Création et configuration d'un compte Atlas

Nous allons créer et configurer ensemble votre compte MongoDB Atlas et Compass.

Mongoose

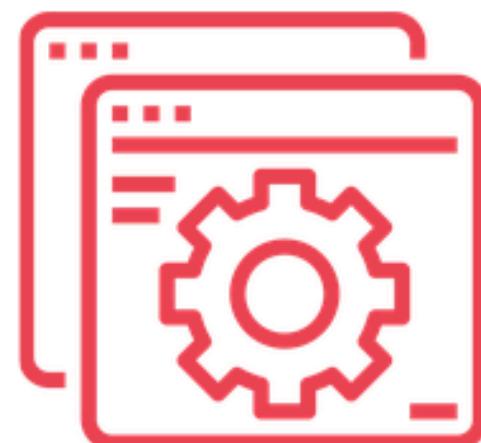


Mongoose

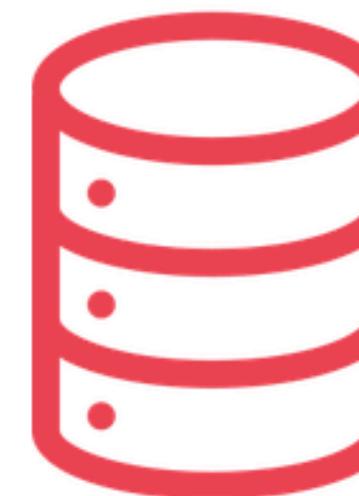
Contexte

Mongoose est un module NodeJS qui permet de faire la passerelle entre une application JavaScript et une base de données.

```
npm install mongoose
```



Application
JavaScript



Base de
données

Mongoose Installation

L'installation de Mongoose est à faire sur chaque nouveau projet pour mettre à jour le package.json.

```
npm install mongoose
```

Mongoose

Fonctionnement

Pour utiliser Mongoose, il faut tout d'abord connecter votre application avec votre base de données grâce à une connection string.

Il s'agit de la même connection string que celle utilisée sur MongoDB Compass.

```
mongodb+srv://admin:password@cluster.mongodb.net/myapp
```

myapp correspond au nom de la base de données à utiliser (un projet = une database).

Attention : il faudra bien penser à changer le nom de la base de données à la fin de l'URL pour chaque nouveau projet.

Mongoose

Communiquer avec MongoDB Atlas

Il est possible de communiquer avec MongoDB grâce à du code javascript et les opérations CRUD.

Les opérations CRUD sont les quatre principales opérations pour manipuler les informations d'une base de données :

- **Create** : ajouter un document
- **Read** : lire le contenu d'une collection
- **Update** : modifier un document
- **Delete** : supprimer un document

Mongoose

Opérations CRUD

Pour illustrer les opérations CRUD, prenez comme exemple la collection ci-dessous.

```
{ _id: "6256a1...", firstName: "John", lastName: "Doe" }  
{ _id: "6256a3...", firstName: "Mike", lastName: "Awesome" }  
{ _id: "6256c0...", firstName: "Maryse", lastName: "Awesome" }
```

Mongoose

Opérations CRUD - récupérer toute une collection

app.js

```
db.find().then(data => {
  console.log(data);
});
```

Résultat

```
[{
  _id: "6256a1...",
  firstName: "John",
  lastName: "Doe",
},
{
  _id: "6256a3...",
  firstName: "Mike",
  lastName: "Awesome",
},
{
  _id: "6256c0...",
  firstName: "Maryse",
  lastName: "Awesome",
}]
```

Mongoose

Opérations CRUD - récupérer plusieurs documents

.find() renverra
toujours un tableau

app.js

```
db.find({ lastName: 'Awesome' })
  .then(data => {
    console.log(data);
  });
}
```

Résultat

```
[{
  _id: "6256a3...",
  firstName: "Mike",
  lastName: "Awesome",
},
{
  _id: "6256c0...",
  firstName: "Maryse",
  lastName: "Awesome",
}]
```

Mongoose

Opérations CRUD - récupérer plusieurs documents

app.js

```
db.find({ lastName: 'Smith' })
  .then(data => {
    console.log(data);
  });

```

Résultat

```
[]
```

Si aucun document n'est trouvé, find() renvoie un **tableau vide**.

Mongoose

Opérations CRUD - récupérer un seul document

app.js

```
db.findOne({ firstName: 'John' })
  .then(data => {
    console.log(data);
  });

```

Résultat

```
{
  _id: "6256a1...",
  firstName: "John",
  lastName: "Doe",
}
```

`findOne()` renverra le document trouvé sous forme d'**objet**.

Mongoose

Opérations CRUD - récupérer un seul document

app.js

```
db.findOne({ firstName: 'Jane' })
  .then(data => {
    console.log(data);
});
```

Résultat

```
null
```

Si aucun document n'est trouvé, findOne() renvoie **null**.

Mongoose

Opérations CRUD - récupérer un seul document via son ID

app.js

```
db.findById('6256a1...')  
  .then(data => {  
    console.log(data);  
  });
```

Résultat

```
{  
  _id: "6256a1..."  
  firstName: "John",  
  lastName: "Doe",  
}
```

findById() renverra le document trouvé
sous forme d'**objet** ou **null** le cas échéant.

Mongoose

Opérations CRUD - modifier un seul document

app.js

```
db.updateOne(
  { firstName: 'John' },
  { lastName: 'Smith' }
).then(() => {
  db.find().then(data =>
  {
    console.log(data);
  });
});
```

Critère de recherche →

Élément à mettre à jour →

Seconde requête pour voir la mise à jour →

Résultat

```
[{
  _id: "6256a1...",
  firstName: "John",
  lastName: "Smith",
},
{
  _id: "6256a3...",
  firstName: "Mike",
  lastName: "Awesome",
},
{
  _id: "6256c0...",
  firstName: "Maryse",
  lastName: "Awesome",
}]
```

Mongoose

Opérations CRUD - modifier plusieurs documents

app.js

```
db.updateMany(  
  { lastName: 'Awesome' },  
  { firstName: 'Mister' }  
).then(() => {  
  
  db.find().then(data => {  
    console.log(data);  
  });  
  
});
```

Résultat

```
[{  
  _id: "6256a1...",  
  firstName: "John",  
  lastName: "Doe",  
},  
,  
{  
  _id: "6256a3...",  
  firstName: "Mister",  
  lastName: "Awesome",  
},  
,  
{  
  _id: "6256c0...",  
  firstName: "Mister",  
  lastName: "Awesome",  
}]
```

Mongoose

Opérations CRUD - supprimer un seul document

app.js

```
db.deleteOne({ firstName: 'John' }).then(() => {  
  db.find().then(data => {  
    console.log(data);  
  });  
});
```

Résultat

```
[{  
  _id: "6256a3...",  
  firstName: "Mike",  
  lastName: "Awesome",  
},  
{  
  _id: "6256c0...",  
  firstName: "Maryse",  
  lastName: "Awesome",  
}]
```

Mongoose

Opérations CRUD - supprimer plusieurs documents

app.js

```
db.deleteMany({ lastName: 'Awesome' }).then(() => {
  db.find().then(data => {
    console.log(data);
  });
});
```

Résultat

```
[{
  _id: "6256a1...",
  firstName: "John",
  lastName: "Smith",
}]
```

Mongoose

Opérations CRUD - créer un document

app.js

1 - Préparation des données →

```
const newUser = new db({  
  firstName: 'Jane',  
  lastName: 'Doe',  
});
```

2 - Sauvegarde des données →

```
newUser.save().then(() => {  
  
  db.find().then(data => {  
    console.log(data);  
  });  
  
});
```

Résultat

```
[{  
  _id: "6256a1...",  
  firstName: "John",  
  lastName: "Smith",  
},  
,  
{  
  _id: "6256a3...",  
  firstName: "Mike",  
  lastName: "Awesome",  
},  
,  
{  
  _id: "6256c0...",  
  firstName: "Maryse",  
  lastName: "Awesome",  
},  
,  
{  
  _id: "6256b1...",  
  firstName: "Jane",  
  lastName: "Doe",  
}]
```

Challenge #1

Play With Mongoose

L'objectif de ce challenge est de vous servir de Mongoose afin de connecter votre application à une base de données et utiliser les opérations CRUD pour importer et manipuler des données externes.



Challenge #1

Play With Mongoose

Faites un git clone de <https://github.com/HedzDev/playwithmongoose.git> et faites un npm install pour installer les dépendances.

👉 Via Compass, créez une base de données “playwithmongoose” et une collection “todos” et importez les données du fichier todos.json à l’intérieur de celle-ci.

💡 Pour importer un fichier JSON dans une collection il vous suffit de vous positionner sur la collection, de cliquer sur le bouton “Add data” > Import file, de sélectionner votre fichier et son format et de valider en cliquant sur “Import”.

👉 Dans le fichier connection.js, complétez la connection string pointant vers la base de données « playwithmongoose ».

👉 Installez le module mongoose.

👉 Dans le fichier app.js, remplissez les différentes fonctions. Leur utilité est précisée dans le commentaire écrit au-dessus de chacune d’elles.

Dans chacune des fonctions (createTodo,completeTodo,deleteTodo), vous devez utiliser l’opération CRUD la plus adaptée (cf le cours).

👉 Appelez les fonctions avec des valeurs de test pour pouvoir vérifier leur résultat en lançant le fichier app.js avec npm start (pour quitter le programme précédemment lancé via npm start faites le raccourci Ctrl + C).

Challenge #2

Scooter Shop - part 1

Le principe du projet ScooterShop est de construire le back office d'un site e-commerce de trottinettes. Vous allez devoir brancher une base de données à votre projet, la remplir et interagir avec celle-ci pour afficher, modifier ou supprimer des données.



Challenge #2

Scooter Shop - part 1

Connexion à la base de données

Faites un git clone de <https://github.com/HedzDev/scootershop-part1.git> et faites un npm install pour installer les dépendances.

👉 Installez le module mongoose.

La première partie de ce projet consiste à créer, compléter puis à connecter votre base de données à votre application.

👉 Via Compass, créez une base de données nommée “scootershop” et, à l’intérieur, une collection “articles”.

👉 Importez le fichier articles.json comportant toutes les données sur les trottinettes à l’intérieur de la collection “articles”.

👉 Dans le fichier connection.js, renseignez votre connection string pointant vers la base de données “scootershop”.

Challenge #2

Scooter Shop - part 1

Manipuler les données

Votre base de données est maintenant connectée et remplie de données, vous allez dès à présent pouvoir les manipuler à travers différentes fonctions.

Chacune de ces fonctions vont vous permettre d'extraire des informations de façon plus ou moins précise. Pour vérifier leur fonctionnement et les données rentrées, testez-les avec les valeurs de votre choix.

Voici l'usage des différentes fonctions:

- `displayAllArticles` : affiche tous les articles de la collection dans votre terminal.
- `displayArticleByName` : affiche un article de la collection dans le terminal en fonction du paramètre "articleName".
- `displayArticleByID` : affiche un article de la collection dans le terminal en fonction du paramètre "articleId" (attention à utiliser la bonne méthode Mongoose).
- `updateArticlePrice` : modifie le prix d'un article dans la base de données en le ciblant avec son id.
- `updateArticleStock` : modifie le stock d'un article dans la base de données en le ciblant avec son id.
- `resetStocks` : réinitialise le stock de tous les articles à 0.

👉 Complétez les 6 fonctions présentes dans le fichier `app.js` en utilisant les bonnes méthodes.

👉 Appelez les fonctions pour pouvoir vérifier leur résultat en lançant le fichier `app.js` avec `npm start`.

Pour quitter un programme précédemment lancé via `npm start` faites le raccourci `Ctrl + C`.

Schéma & Modèle



Schéma & Modèle

Contexte

Une fois la connexion établie entre la base de données et l'application, Mongoose a obligatoirement besoin de deux éléments pour manipuler cette base de données : un schéma et un modèle.



Un schéma

+



Un modèle

Schéma & Modèle

Fonctionnement

Le schéma décrit de façon précise les données qui vont être manipulées pour les documents de chaque collection.

```
const todoSchema = mongoose.Schema({
  name: String,
  assignedTo: [String],
  priority: Number,
  done: Boolean,
  dateDue: Date,
});
```

Dans cet exemple, le schéma s'appelle "todoSchema" et il possède cinq données différentes. Il faut préciser le type de chacune d'elles.

Schéma & Modèle

Fonctionnement

Le modèle permet d'enregistrer le schéma auprès d'une collection spécifique.

```
const Todo = mongoose.model('todos', todoSchema);
```

Ici, le schéma est enregistré pour la collection "todos".

Attention : le nom de la collection doit obligatoirement être au pluriel.

Schéma & Modèle

Le modèle et le schéma

models/todos.js

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

const todoSchema = mongoose.Schema({
  name: String,
  assignedTo: [String],
  priority: Number,
  done: Boolean,
  dateDue: Date,
});

const Todo = mongoose.model('todos', todoSchema);

module.exports = Todo;
```

Export du modèle afin
de l'utiliser dans l'application

Schéma & Modèle

Connexion à la BDD

Cette partie va illustrer comment connecter une collection "todos" à une application Node JS afin de pouvoir la manipuler.

👉 Dans l'avenir vous pourrez simplement récupérer cette structure en l'adaptant à votre base de données et à vos collections.

models/connection.js

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

const connectionString = 'mongodb+srv://.../todoapp'; ←

mongoose.connect(connectionString, { connectTimeoutMS: 2000 })
  .then(() => console.log('Database connected'))
  .catch(error => console.error(error));
```

Nom de la base de données
Un projet = une database

Schéma & Modèle

Exemple d'utilisation

app.js

```
1- Initialisation de la connexion → require('./models/connection');

2- Import du modèle → const Todo = require('./models/todos');

3- Utilisation du modèle → Todo.find().then(data => { console.log(data); });

const newTodo = new Todo({
  name: 'Walk the dog',
  assignedTo: ['John', 'Jane'],
  priority: 1,
  done: false,
  dateDue: new Date(),
});

newTodo.save().then(newDoc => {
  console.log(newDoc);
});
```

À partir de maintenant vous n'utiliserez plus la variable "db" mais la variable que vous aurez assigné à chacun de vos modèles.

Challenge #3

Scooter Shop - part 2

Lors de cette seconde partie, vous allez récupérer les fonctions que vous avez codées sur la partie 1 mais vous devrez importer une nouvelle base de données en la connectant à votre application et en écrivant les schémas et les modèles vous-même.

L'objectif sera de vérifier que vos fonctions se lancent toujours correctement et d'en coder d'autres vous permettant de manipuler les nouvelles collections ajoutées.

N'hésitez pas à supprimer votre ancienne BDD scootershop pour davantage de clarté.



Challenge #3

Scooter Shop - part 2

Importer la nouvelle base de données

Faites un git clone de <https://github.com/HedzDev/scootershop-part2.git> et faites un npm install pour installer les dépendances.

- 👉 Importez via compass les fichiers articles.json, users.json et orders.json à l'intérieur de votre base de données scootershop (le nom des collections sera: articles, users et orders).

Challenge #3

Scooter Shop - part 2

Connexion de la nouvelle base de données

- 👉 Dans votre application (obligatoirement dans le dossier `models`, à la racine du projet), créez les 3 nouveaux schémas et modèles correspondants aux collections nouvellement importées (créez un fichier par schéma/modèle).
- 👉 Toujours dans le dossier `models`, ajoutez le fichier de connexion à votre base de donnée comme vu dans le cours.
- 👉 Lancez votre application avec la commande “`npm start`” pour vérifier que vos fonctions créées hier s'exécutent toujours correctement.

Challenge #3

Scooter Shop - part 2

Création de nouvelles fonctions

Votre base de données est maintenant à nouveau connectée et celle-ci est complétée de deux nouvelles collections. Comme hier, vous allez dès à présent pouvoir coder de nouvelles fonctions pour manipuler ces collections.

Pour vérifier le fonctionnement de chaque fonction et des données qu'elles retournent, testez-les avec les valeurs de votre choix.

Voici l'usage des nouvelles fonctions :

- `displayAllUsers` : doit afficher la liste de tous les utilisateurs de la collection `users` dans votre terminal.
- `deleteUser` : doit supprimer un utilisateur en ciblant son ID.
- `displayAllOrders` : doit afficher la liste de toutes les commandes de la collection `orders` dans votre terminal.
- `updateOrderPayment` : doit mettre à jour le statut de paiement (`paid: true`) d'une commande en ciblant son ID.
- `deleteOrder` : doit supprimer une commande en ciblant son ID.

👉 Complétez les 5 fonctions présentes dans le fichier `app.js` en utilisant les bonnes méthodes.

👉 Appelez les fonctions pour pouvoir vérifier leur résultat en lançant le fichier `app.js` avec `npm start`.

Pour quitter un programme précédemment lancé via `npm start` faites le raccourci `Ctrl + C`.

Modélisation & Relations



Modélisation & Relations

Contexte

La modélisation, à quoi ça sert?

La base de données représente les fondations du projet, il est donc nécessaire de structurer les données enregistrées.

Cette structuration permet de prévoir l'évolution des fonctionnalités de l'application, d'optimiser leur traitement et de dresser les contraintes du projet à travers les données enregistrées.

La modélisation est rarement effectuée par des développeurs juniors mais il est important de se familiariser avec cette pratique.

Modélisation & Relations

Fonctionnement

La modélisation consiste à faire un état des lieux d'une partie de l'application, le but étant de dresser une maquette listant et organisant l'ensemble des informations nécessaires à celle-ci.

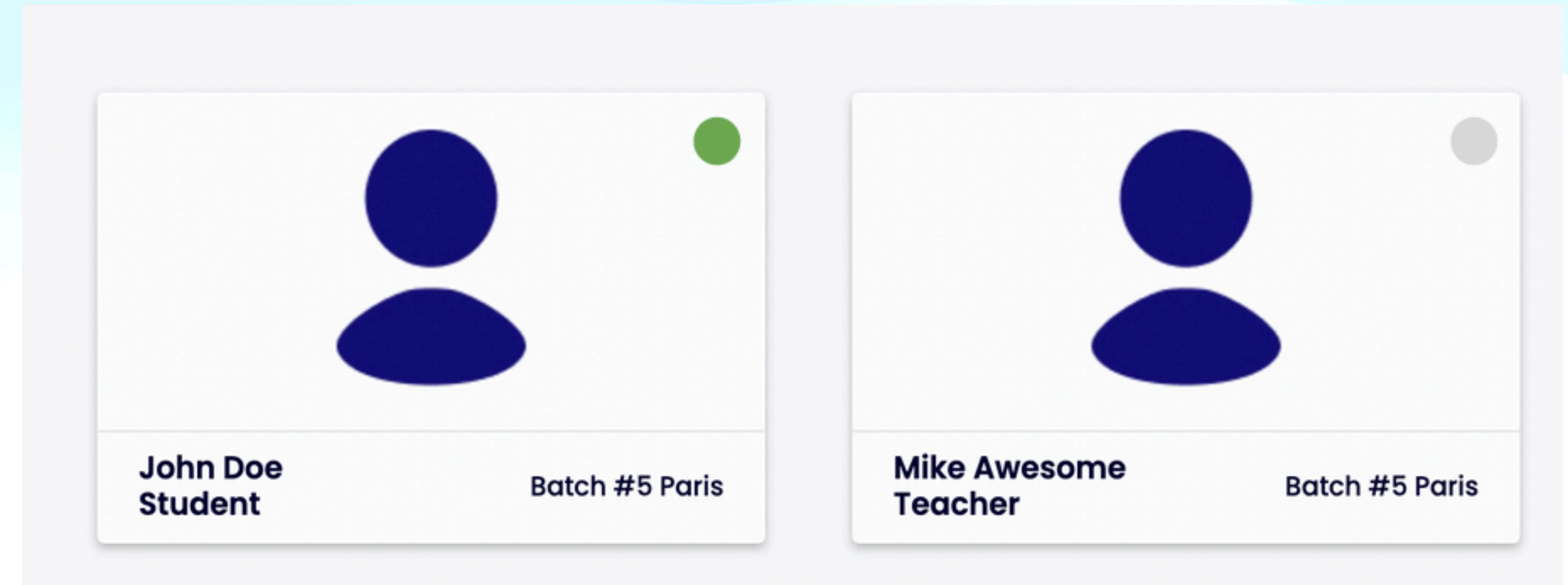
La première étape est de constituer la liste des données utiles.

Modélisation & Relations

Fonctionnement

Dans l'exemple ci-dessous, la liste des informations nécessaires est:

- Prénom
- Nom
- Rôle
- En ligne
- Ville
- Numéro de batch



Modélisation & Relations

Fonctionnement

Une fois les éléments listés il est important de les typer dans le but d'appréhender la construction du schéma de base de données.

- Prénom : String
- Nom : String
- Rôle : String
- En ligne : Booléens
- Ville : String
- Numéro de batch : Number

Modélisation & Relations

Fonctionnement

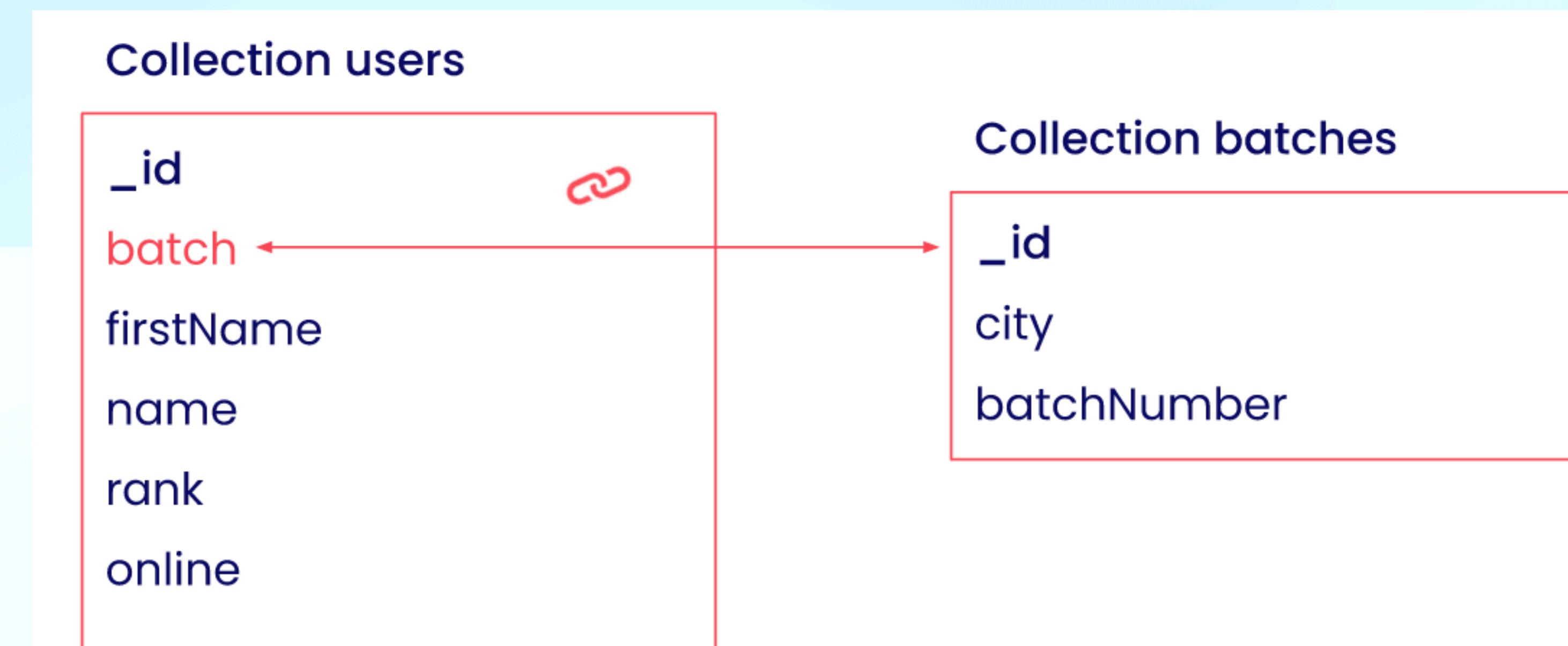
La seconde étape consiste à regrouper ces informations, c'est là qu'intervient le principe de relations.

Il existe généralement deux façons pour gérer les relations entre les informations: les **sous-documents** et les **clés étrangères**.

Modélisation & Relations

Fonctionnement

La clé étrangère consiste à lier deux documents distincts via une clé (un id).



Dans cet exemple la clé sera l'id du batch

Modélisation & Relations

Fonctionnement

Le sous-document quant à lui permet de fusionner ces deux collections en une seule.



Modélisation & Relations

Comment choisir entre une clé étrangère et un sous-document ?

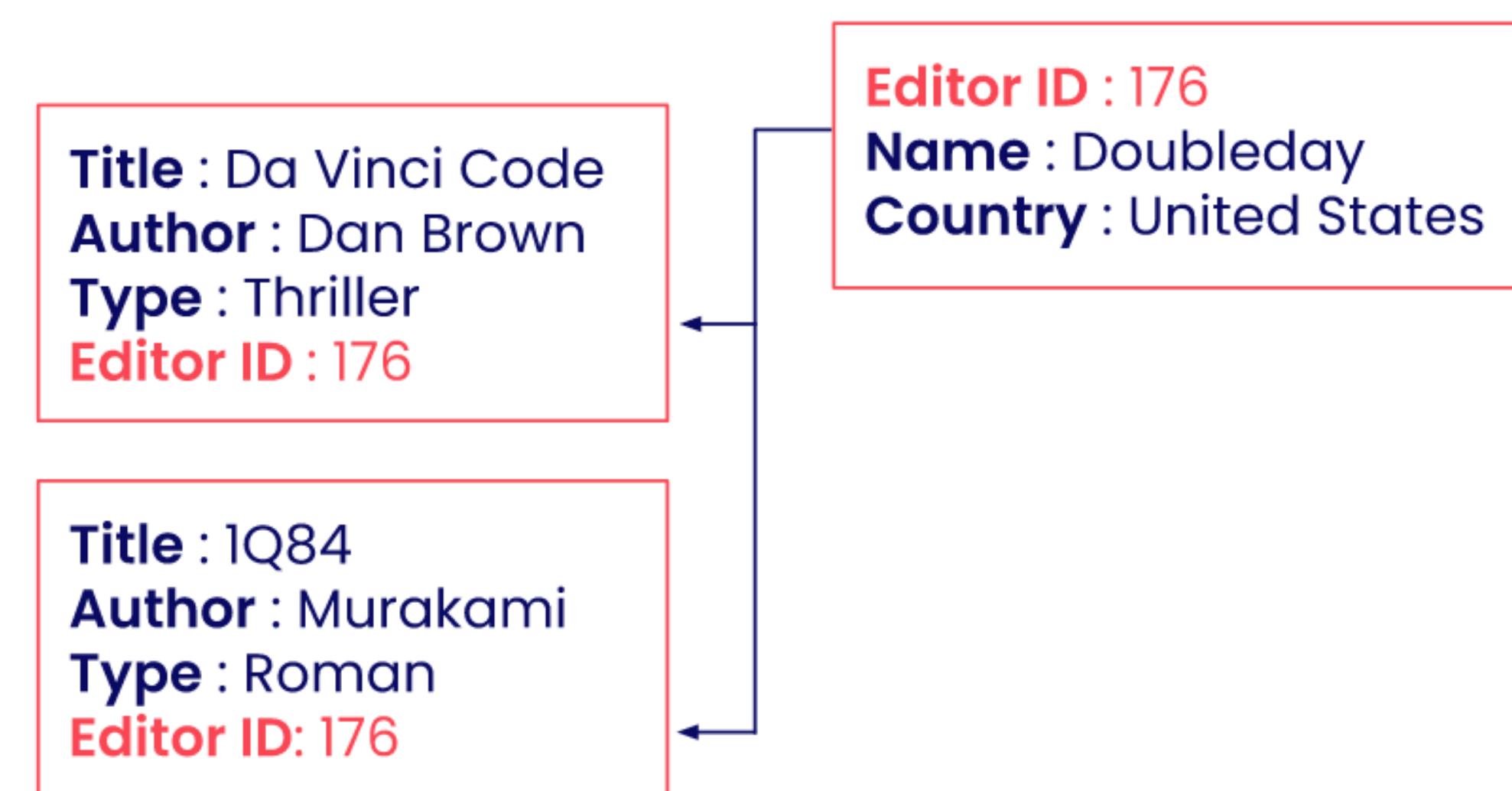
Pour savoir quelle solution est la plus optimale, il faut comprendre l'usage de vos données.

Les informations à relier sont-elles par exemple utiles à un seul utilisateur ou à tous les utilisateurs de l'application ?

Modélisation & Relations

Contexte optimal pour les clés étrangères

Dans ce contexte, chaque livre est relié à un éditeur.



Un éditeur se retrouvera forcément sur plusieurs livres.

Il est donc plus optimal de détacher cette information dans un document à part.

Modélisation & Relations

Contexte optimal pour les sous-documents

Dans ce contexte, chaque adresse est reliée à un utilisateur.

Lastname : John
Firstname : Doe
Email : john@gmail.com

Adresses :

1 **Street** : 2 Rue des Lilas
City : Paris

2 **Street** : 5 Rue Papin
City : Paris

Il est peu probable qu'un autre utilisateur partage l'adresse d'un autre utilisateur.

Il est donc plus pertinent d'utiliser la mécanique des sous-documents.

Modélisation & Relations

Les clés étrangères

Les clés étrangères amènent un nouveau type de données : **ObjectId**.

Les extraits de codes ci-après se basent sur l'exemple des livres et des éditeurs.
Attention il faudrait ajouter les imports/exports pour pouvoir les utiliser correctement.

Modélisation & Relations

Les clés étrangères

Préparation de la relation

models/books.js

```
const bookSchema = mongoose.Schema({
  title: String,
  author: String,
  type: String,
  editor: { type: mongoose.Schema.Types.ObjectId, ref: 'editors' },
});

const Book = mongoose.model('books', bookSchema);
```

models/editors.js

```
const editorSchema = mongoose.Schema({
  name: String,
  country: String,
});

const Editor = mongoose.model('editors', editorSchema);
```

Modélisation & Relations

Les clés étrangères

Création de la relation

app.js

```
const newBook = new Book({
  title: 'Da Vinci Code',
  author: 'Dan Brown',
  type: 'thriller',
  editor: '6256a1...', ← _id d'un éditeur
});

newBook.save().then(() => {
  console.log('Book saved!');
});
```

Modélisation & Relations

Les clés étrangères

L'instruction populate

En pratique, l'instruction **populate** permettra d'exploiter la relation afin de récupérer le document lié sous forme d'objet.

app.js

Populate avec le nom du champ contenant l'ObjectId

```
Book.findOne({ name: 'Da Vinci Code' })
  .populate('editor')
  .then(data => {
    console.log(data);
  });

```

Résultat

```
{
  name: "Da Vinci Code",
  author: "Dan Brown",
  type: "thriller",
  editor: {
    name: "Doubleday",
    country: "US"
  }
}
```

Modélisation & Relations

Les clés étrangères

Lier plusieurs documents

Il est possible de lier un livre à plusieurs éditeurs grâce à un tableau de clés étrangères.

models/books.js

```
const bookSchema = mongoose.Schema({
  title: String,
  author: String,
  type: String,
  editors: [{ type: mongoose.Schema.Types.ObjectId, ref: 'editors' }],
});

const Book = mongoose.model('books', bookSchema);
```

Modélisation & Relations

Les clés étrangères

Lier plusieurs documents

Lors de la création de la référence, il sera donc possible de spécifier plusieurs ID sous la forme d'un tableau.

app.js

```
const newBook = new Book({
  title: 'Da Vinci Code',
  author: 'Dan Brown',
  type: 'thriller',
  editors: ['6256a1...', '6256c0...'],
});

newBook.save().then(() => {
  console.log('Book saved!');
});
```

Modélisation & Relations

Les clés étrangères

Populate sur un tableau d'ObjectId

Même mécanique, mais le résultat sera retourné sous la forme d'un tableau d'objets.

app.js

```
Book.findOne({ name: 'Da Vinci Code' })
  .populate('editors')
  .then(data => {
    console.log(data);
 });
```

Résultat

```
{
  name: "Da Vinci Code",
  author: "Dan Brown",
  type: "thriller",
  editors: [
    { name: "Doubleday", country: "US" },
    { name: "Tripleday", country: "UK" }
  ]
}
```

Modélisation & Relations

Les sous-documents

Le sous-document s'apparente à un schéma dans un schéma où la valeur d'une des propriétés du schéma principal fait référence au second.

Les extraits de codes ci-après se basent sur l'exemple d'un utilisateur et de ses adresses. Attention il faudrait ajouter les imports/exports pour pouvoir les utiliser correctement.

Modélisation & Relations

Les sous-documents

Un schéma dans un schéma

models/users.js

```
const addressSchema = mongoose.Schema({
  street: String,
  city: String,
});

const userSchema = mongoose.Schema({
  firstName: String,
  lastName: String,
  email: String,
  address: addressSchema,
});

const User = mongoose.model('users', userSchema);
```

Modélisation & Relations

Les sous-documents

Écrire un sous-document

app.js

```
const newUser = new User({
  firstName: 'John',
  lastName: 'Doe',
  email: 'john@gmail.com',
  address: {
    street: '2 Rue des Lilas',
    city: 'Paris',
  },
});

newUser.save().then(() => {
  console.log('User saved!');
});
```

Modélisation & Relations

Les sous-documents

Lire un sous-document

app.js

```
User.findOne({ firstName: 'John' })
  .then(data => {
    console.log(data);
  });

```

Résultat

```
{
  firstName: "John",
  lastName: "Doe",
  email: "john@gmail.com",
  address: {
    street: "2 Rue des Lilas",
    city: "Paris"
  }
}
```

Modélisation & Relations

Les sous-documents

Un tableau de sous-documents

Il est possible d'enregistrer plusieurs adresses pour un utilisateur grâce à un tableau de sous-documents.

models/users.js

```
const addressSchema = mongoose.Schema({
  street: String,
  city: String,
});

const userSchema = mongoose.Schema({
  firstName: String,
  lastName: String,
  email: String,
  addresses: [addressSchema],
});

const User = mongoose.model('users', userSchema);
```

Modélisation & Relations

Les sous-documents

Écrire un tableau de sous-documents

app.js

```
const newUser = new User({
  firstName: 'John',
  lastName: 'Doe',
  email: 'john@gmail.com',
  addresses: [
    { street: '2 Rue des Lilas', city: 'Paris' },
    { street: '5 Rue Papin', city: 'Paris' },
  ],
});

newUser.save().then(() => {
  console.log('User saved!');
});
```

Modélisation & Relations

Les sous-documents

Lire un tableau de sous-documents

app.js

```
User.findOne({ firstName: 'John' })
  .then(data => {
    console.log(data);
  });

```

Résultat

```
{
  firstName: "John",
  lastName: "Doe",
  email: "john@gmail.com",
  addresses: [
    { street: "2 Rue des Lilas", city: "Paris" },
    { street: "5 Rue Papin", city: "Paris" }
  ]
}
```

Challenge #4

Scooter Shop - part 3

Lors de cette troisième partie, vous allez à nouveau récupérer les fonctions que vous avez codées les jours précédents mais certaines collections ont subi des modifications...

Il faudra vous adapter à ces changements pour que vos anciennes requêtes fonctionnent toujours correctement et vous devrez en développer des nouvelles, plus complexes.



Challenge #4

Scooter Shop - part 3

Importer la nouvelle base de données

Faites un git clone de <https://github.com/HedzDev/scootershop-part3.git> et faites un npm install pour installer les dépendances.

- 👉 Importez via compass les fichiers articles.json, users.json et orders.json à l'intérieur de votre base de données scooter-shop (le nom des collections sera: articles, users et orders).

Challenge #4

Scooter Shop - part 3

[Modifier les schémas](#)

- 👉 Modifiez les schémas des collections users et orders pour qu'ils s'adaptent à leur nouvelle structure.
- 💡 Il faudra notamment faire évoluer le schéma de la collection users en y incluant un tableau de sous-documents chargé d'enregistrer les différentes adresses.
- 👉 Lancez votre application avec la commande “npm start” pour vérifier que vos fonctions créées les jours précédents s'exécutent toujours correctement.

Challenge #4

Scooter Shop - part 3

Création de nouvelles fonctions

👉 Complétez les 2 nouvelles fonctions présentes en bas du fichier app.js en utilisant les bonnes méthodes.

Voici l'usage de ces fonctions :

- displayOrderArticles (by orderId with populate): doit afficher dans votre terminal la liste des articles d'une commande ciblée par son id.

- displayUserOrders (by userId): doit afficher dans votre terminal la liste des commandes d'un client ciblée par son id.

👉 Appelez les fonctions pour pouvoir vérifier leur résultat en lançant le fichier app.js avec npm start.

Challenge #4

Scooter Shop - part 3

Visualiser des données dans la console

Il est possible d'utiliser des outils pour visualiser des données dans la console comme par exemple [Ervy](#).

Imaginons que nous voulions représenter le stock de tous nos produits dans la console...

Challenge #4

Scooter Shop - part 3

Visualiser les données dans la console

```
1  require('../models/connection');
2  const { bg, bullet } = require('ervy');
3  const Article = require('../models/articles');
4  const User = require('../models/users');
5  const Order = require('../models/orders');
6
7  /*
8   ** Articles
9   */
10
11 function displayAllArticles() {
12   Article.find().then((data) => {
13     const dataTable = [];
14     for (const article of data) {
15       dataTable.push({
16         key: article.name,
17         value: article.stock,
18         style: article.stock > 5 ? bg('blue') : bg('red'),
19         height: 30,
20       });
21     }
22     console.log(bullet(dataTable));
23   });
24 }
25 displayAllArticles();
```



Démarrer avec Express



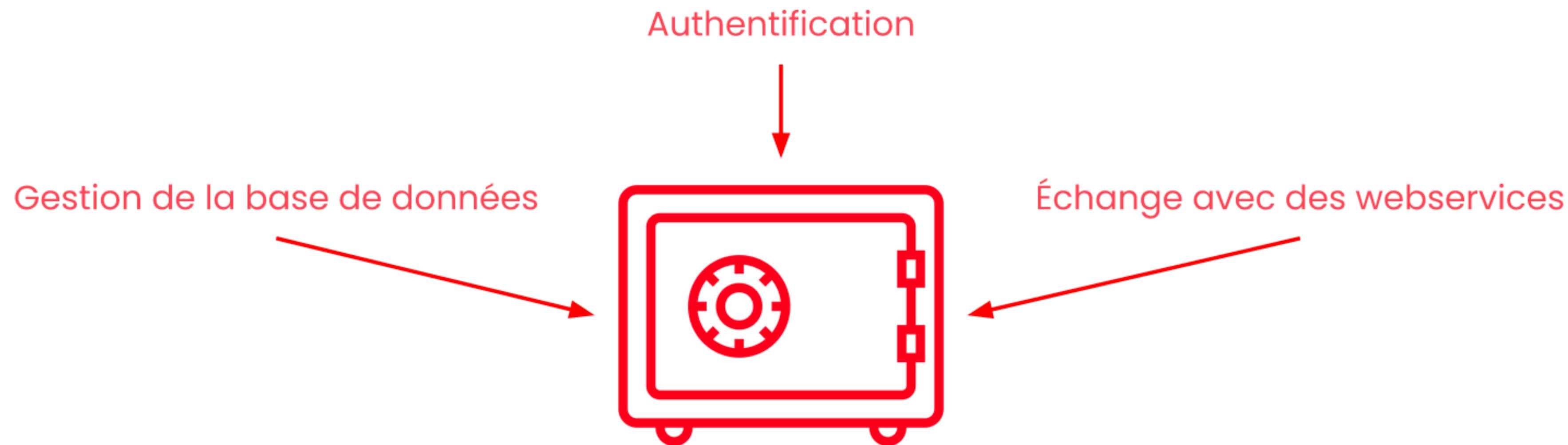
Express  JS

The text "Express" is in a black sans-serif font. To its right is a yellow square containing the letters "JS" in a bold black font. A registered trademark symbol (®) is located at the bottom right of the "JS" logo.

Démarrer avec Express

Intérêt du backend

L'accès via le navigateur à certaines données présente **des risques de sécurité**. Le traitement de ces données par un backend permet de les **protéger**.

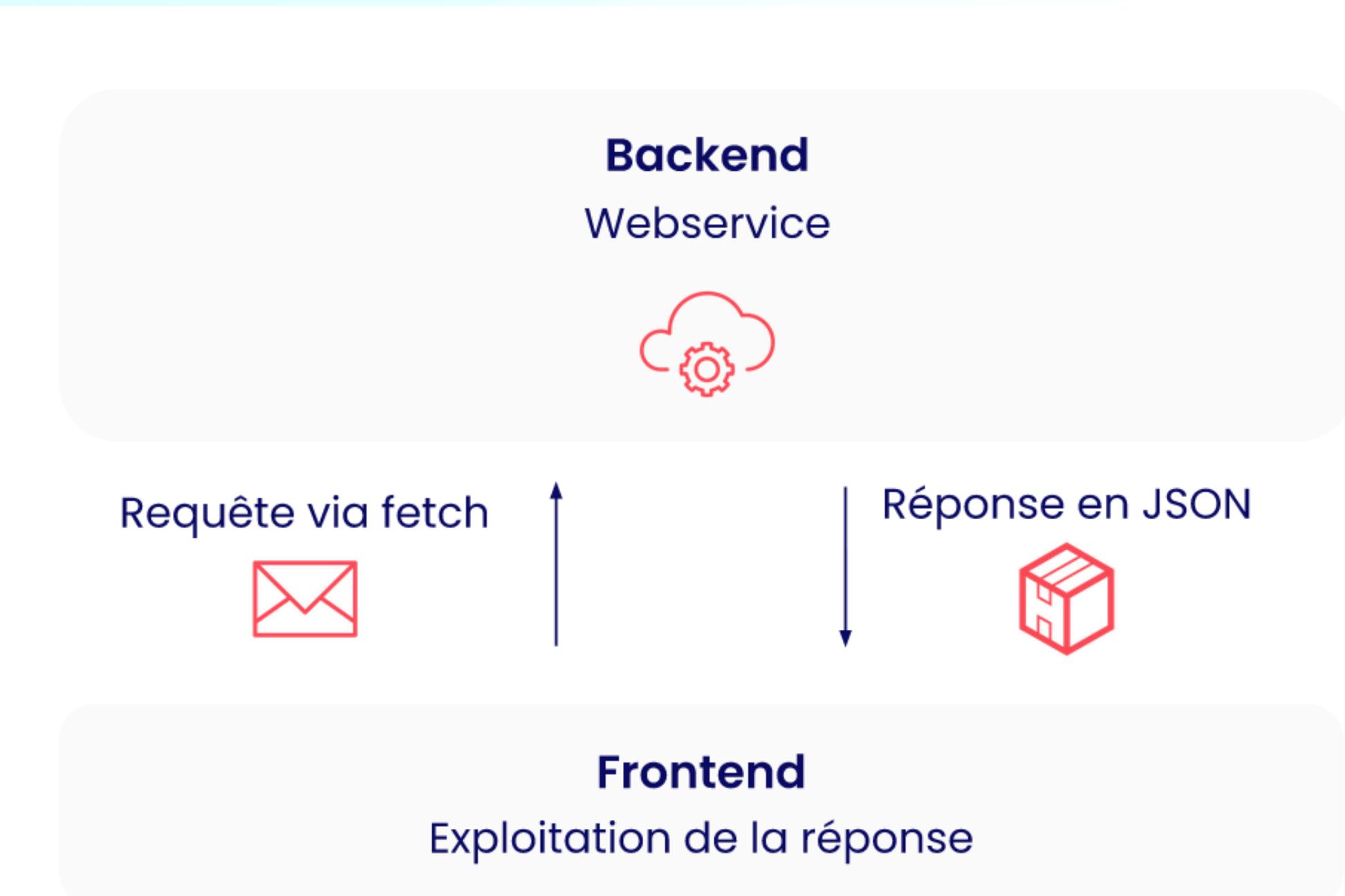


Démarrer avec Express

Un webservice comme backend

Le framework Express facilite la création d'un backend qui jouera le rôle du webservice.

La création d'un backend sous forme d'un webservice permet **l'échange de données avec le frontend**.

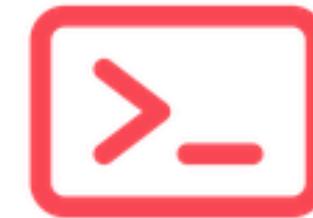


Démarrer avec Express

Fonctionnement

Installation du générateur de backend

À ne faire **qu'une seule fois** sur votre ordinateur.



```
npm install express-generator -g
```



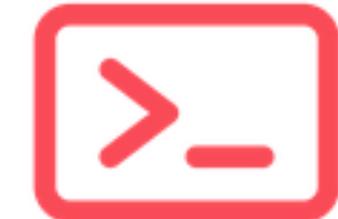
Vous devez ajouter **sudo** devant cette commande
si vous êtes sous MacOS ou Linux.

Démarrer avec Express

Fonctionnement

Génération du backend

La seconde étape est de générer un backend avec la structure propre à Express à l'intérieur d'un dossier (en précisant le répertoire courant avec `./`).



```
express --no-view --git ./
```

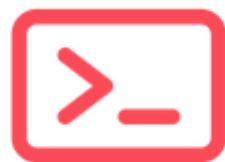
Puis finaliser l'initialisation d'Express dans votre projet en installant les dépendances en tapant **npm install**

Démarrer avec Express

Fonctionnement

Paramétrage du webservice

Il faut ajouter le module CORS afin que notre webservice puisse être joint par une application JS.



`npm install cors`

Démarrer avec Express

Fonctionnement

Paramétrage du webservice

Nous ajoutons le module **CORS** afin que notre webservice puisse être joint par une application JS par la suite.

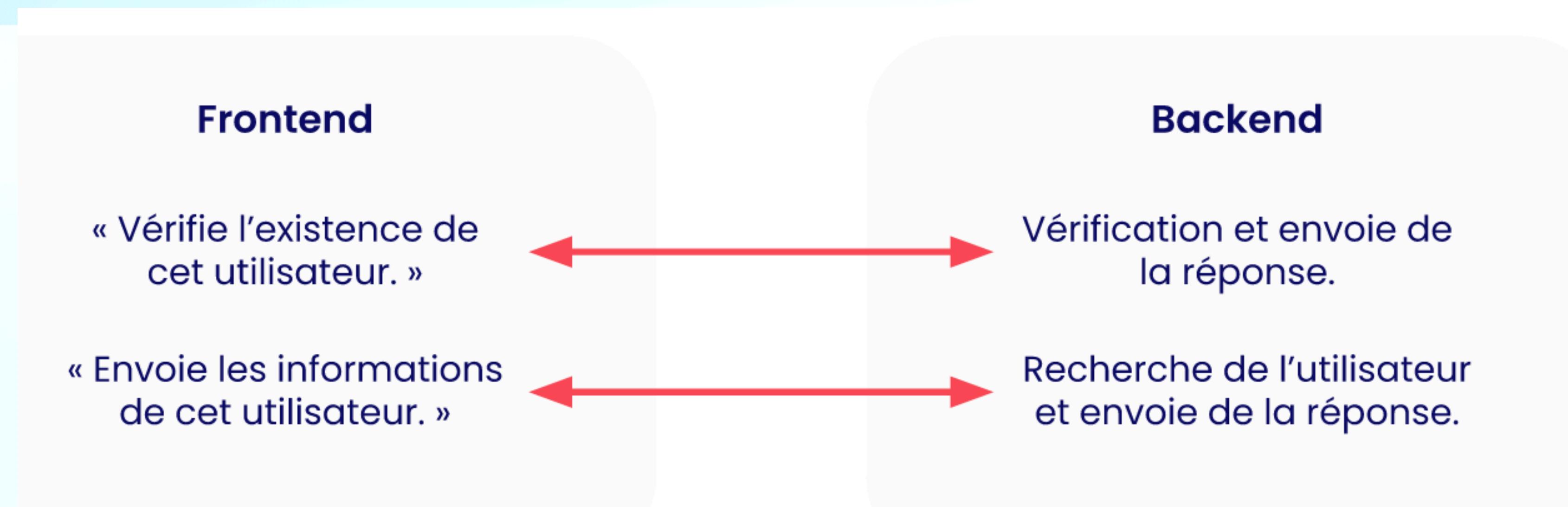
app.js - à ajouter ligne 10

```
...
const cors = require('cors');
app.use(cors());
...
...
```

Démarrer avec Express

Les routes

Les routes permettent au backend d'associer une demande à un traitement précis.



Démarrer avec Express

Les routes

Le répertoire “routes” comporte les fichiers des routes d'un projet.

Le fichier routes/index.js est le centre névralgique de votre webservice.

```
> 📁 bin  
> 📂 public  
└(routes  
    └── index.js  
    └── users.js  
    └── .gitignore  
    └── app.js  
    └── package.json
```

Démarrer avec Express

Les routes

Création de votre première route

routes/index.js

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

router.get('/date', (req, res) => {
  const date = new Date();
  res.json({ now: date });
});

module.exports = router;
```

Démarrer avec Express

Les routes

Structure d'une route



Démarrer avec Express

Les routes

Test d'une route avec Thunder Client

Pour tester facilement vos routes, vous pouvez utiliser l'extension Thunder Client depuis VSCode.

⚠ Relancer le serveur avant de tester

URL du backend
`http://localhost:3000/nomDeLaRoute`

Type de requête → GET http://localhost:3000/date Send

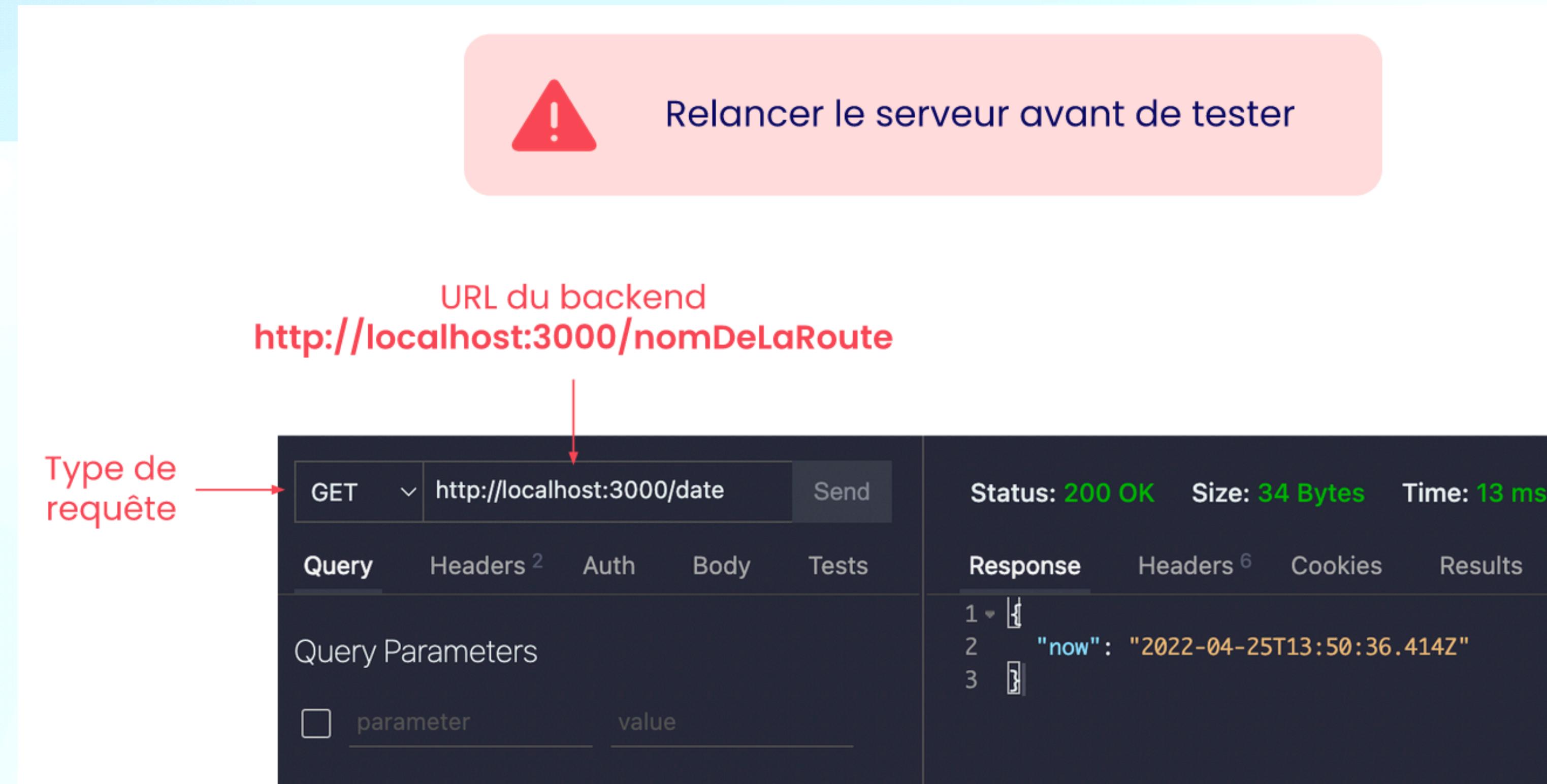
Query Headers² Auth Body Tests

Query Parameters

1 []
2 "now": "2022-04-25T13:50:36.414Z"
3 []

Status: 200 OK Size: 34 Bytes Time: 13 ms

Response Headers⁶ Cookies Results



Démarrer avec Express

Les opérations CRUD

Pour rappel le CRUD correspond à toutes les opérations que l'on va pouvoir mettre en place via notre webservice : Create, Read, Update et Delete.

Dans les exemples ci-après, le rôle du backend en webservice sera de gérer et manipuler une liste d'artistes.

Démarrer avec Express

Les opérations CRUD

Récupérer les artistes

Nous utiliserons une variable globale pour stocker cette liste.

```
let artists = ['Daft Punk', 'The Beatles'];

router.get('/artists', (req, res) => {
  res.json({ artistsList: artists });
});
```

Démarrer avec Express

Les opérations CRUD

Tester une requête GET

The screenshot shows the POSTMAN application interface. On the left, the request configuration is displayed: a GET method, the URL `http://localhost:3000/artists`, and a 'Send' button. Below this, tabs for 'Query', 'Headers', 'Auth', 'Body', and 'Tests' are visible, with 'Query' being the active tab. Under 'Query Parameters', there is a table with one row containing a checkbox labeled 'parameter' and a text input labeled 'value'. On the right, the response details are shown: 'Status: 200 OK', 'Size: 43 Bytes', and 'Time: 19 ms'. Below these, tabs for 'Response', 'Headers', 'Cookies', 'Results', and 'Docs' are present, with 'Response' being the active tab. The response body is a JSON object: `{"artistsList": ["Daft Punk", "The Beatles"]}`. The JSON structure is highlighted with a red box around the array element at index 2.

Démarrer avec Express

Les opérations CRUD

Ajouter un artiste

```
let artists = ['Daft Punk', 'The Beatles'];

router.post('/artists', (req, res) => {
  artists.push(req.body.newArtist);
  res.json({ artistsList: artists });
});
```

Ajout d'un nouvel artiste
via la méthode *push*

Démarrer avec Express

Les opérations CRUD

Tester une requête POST

The screenshot shows the Postman interface. In the top left, there's a POST request to `http://localhost:3000/artists`. The 'Body' tab is selected, and under it, 'Form-encode' is chosen. The body contains two fields: 'newArtist' with a checked checkbox and 'name' with the value 'Ed Sheeran'. The response tab shows a status of 200 OK, size of 56 Bytes, and time of 18 ms. The response body is a JSON object with an 'artistsList' key containing three values: 'Daft Punk', 'The Beatles', and 'Ed Sheeran'.

```
1 [{}  
2 "artistsList": [  
3   "Daft Punk",  
4   "The Beatles",  
5   "Ed Sheeran"  
6 ]  
7 ]
```

Envoi du nouvel artiste au format clé / valeur
L'information sera disponible dans `req.body.newArtist`

Démarrer avec Express

Les opérations CRUD

Mettre à jour un artiste

Il est possible de préparer l'URL à recevoir un paramètre sous forme de **params**.

```
let artists = ['Daft Punk', 'The Beatles'];

router.put('/artists/:position', (req, res) => {
  artists[req.params.position] = req.body.replacementArtist;
  res.json({ artistsList: artists });
});
```

Dans cet exemple, le params envoyé est **:position**, il correspond aux params envoyés à la position de l'élément à mettre à jour.

Démarrer avec Express

Les opérations CRUD

Tester une requête PUT

Information disponible dans `req.params.position`

The screenshot shows a POSTMAN interface. The URL is `http://localhost:3000/artists/1`. The body is set to `Form-encode` and contains a field `replacementArtist` with value `Eminem`. The response status is `200 OK`, size `38 Bytes`, and time `19 ms`. The response body is a JSON object with a single key `artistsList` containing an array of two elements: `"Daft Punk"` and `"Eminem"`.

Information disponible dans `req.body.replacementArtist`

Démarrer avec Express

Les opérations CRUD

Supprimer un artiste

```
let artists = ['Daft Punk', 'The Beatles'];

router.delete('/artists/:position', (req, res) => {
  artists.splice(req.params.position, 1); ←
  res.json({ artistsList: artists });
});
```

Retrait d'un artiste
via la méthode *splice*

Démarrer avec Express

Les opérations CRUD

Tester une requête DELETE

The screenshot shows the POSTMAN application interface. On the left, the request details are set to a **DELETE** method at **http://localhost:3000/artists/0**. The response status is **200 OK**, size is **31 Bytes**, and time is **9 ms**. The response body is displayed as JSON:

```
1 [{}  
2 "artistsList": [  
3     "The Beatles"  
4 ]  
5 ]
```

Démarrer avec Express

Les opérations CRUD

Le retour d'une requête

Préciser le retour d'une requête fait partie des bonnes pratiques à garder en tête. Cela permet d'avoir des informations sur le déroulement de celle-ci comme par exemple le statut.

Si aucun artiste n'est reçu, le retour renverra la propriété "result" à false indiquant ainsi que le traitement s'est mal passé.

```
router.post('/artists', (req, res) => {
  if (req.body.newArtist) {
    artists.push(req.body.newArtist);
    res.json({ result: true, artistsList: artists });
  } else {
    res.json({ result: false });
  }
});
```

Challenge #5

WeatherApp - part 1

WeatherApp est une application qui affiche la météo des villes que vous aurez saisies.

La première partie se concentre sur la construction du backend en webservice chargé de faire transiter les informations météorologiques.

Vous n'aurez pour le moment pas besoin d'API externe, toutes les données sont stockées dans la variable globale nommée `weather`.



Challenge #5

WeatherApp - part 1

Faites un git clone de <https://github.com/HedzDev/weatherapp-part1> et faites un npm install pour installer les dépendances.

Challenge #5

WeatherApp - part 1

Création de vos routes

👉 Dans le fichier routes/index.js créez, dans l'ordre, les routes suivantes avec des retours de requêtes pour gérer les cas d'erreurs ou de doublons :

- POST /weather : ajoute une nouvelle ville avec sa météo et retourne uniquement les informations de la nouvelle ville créée.

Faites en sorte que votre réponse soit personnalisée en cas d'ajout de doublon et pensez à gérer la casse.

💡 Les retours de requêtes vous permettent de personnaliser vos résultats selon des contraintes conditionnelles (si ma ville n'existe pas je l'ajoute et je la retourne sinon, je renvoie un message d'erreur).

💡 Pour savoir si une ville existe déjà dans le tableau weather, vous pouvez utiliser la méthode .some() ou .find().

💡 La gestion de la casse permettra par exemple à votre code de considérer “London”, “london” ou “LONDON” comme la même ville est ainsi éviter des doublons liés aux majuscules et minuscules.

Pour gérer la casse vous pouvez utiliser la méthode .toLowerCase().

Challenge #5

WeatherApp - part 1

Création de vos routes

Voici un exemple de route avec un retour de requête :

```
router.post('/artists', (req, res) => {
  if (req.body.newArtist) {
    artists.push(req.body.newArtist);
    res.json({ result: true, artistsList: artists });
  } else {
    res.json({ result: false, error: "no artist filled" });
  }
});
```

Challenge #5

WeatherApp - part 1

Création de vos routes

- GET /weather : renvoie toutes les données du tableau weather.
- GET /weather/:cityName : renvoie les informations météos en fonction d'une ville ciblée.

Gérez votre réponse de requête dans le cas où la ville ciblée n'existe pas dans le tableau weather et pensez à gérer la casse.

La valeur que prend un params ne doit pas être écrite en dur dans votre route, elle sera renseignée à la fin de l'url que vous testerez sur Thunder Client.

Challenge #5

WeatherApp - part 1

Création de vos routes

- `DELETE /weather/:cityName` : supprime une ville ciblée dans le tableau `weather`.

Gérez votre réponse de requête dans le cas où la ville ciblée n'existe pas dans le tableau `weather` et pensez à gérer la casse.

👉 Vérifiez vos routes en les testant avec l'extension `ThunderClient`. Si besoin n'oubliez pas de redémarrer votre projet avec la commande “`yarn start`”.

Les services tiers



Les services tiers

Contexte

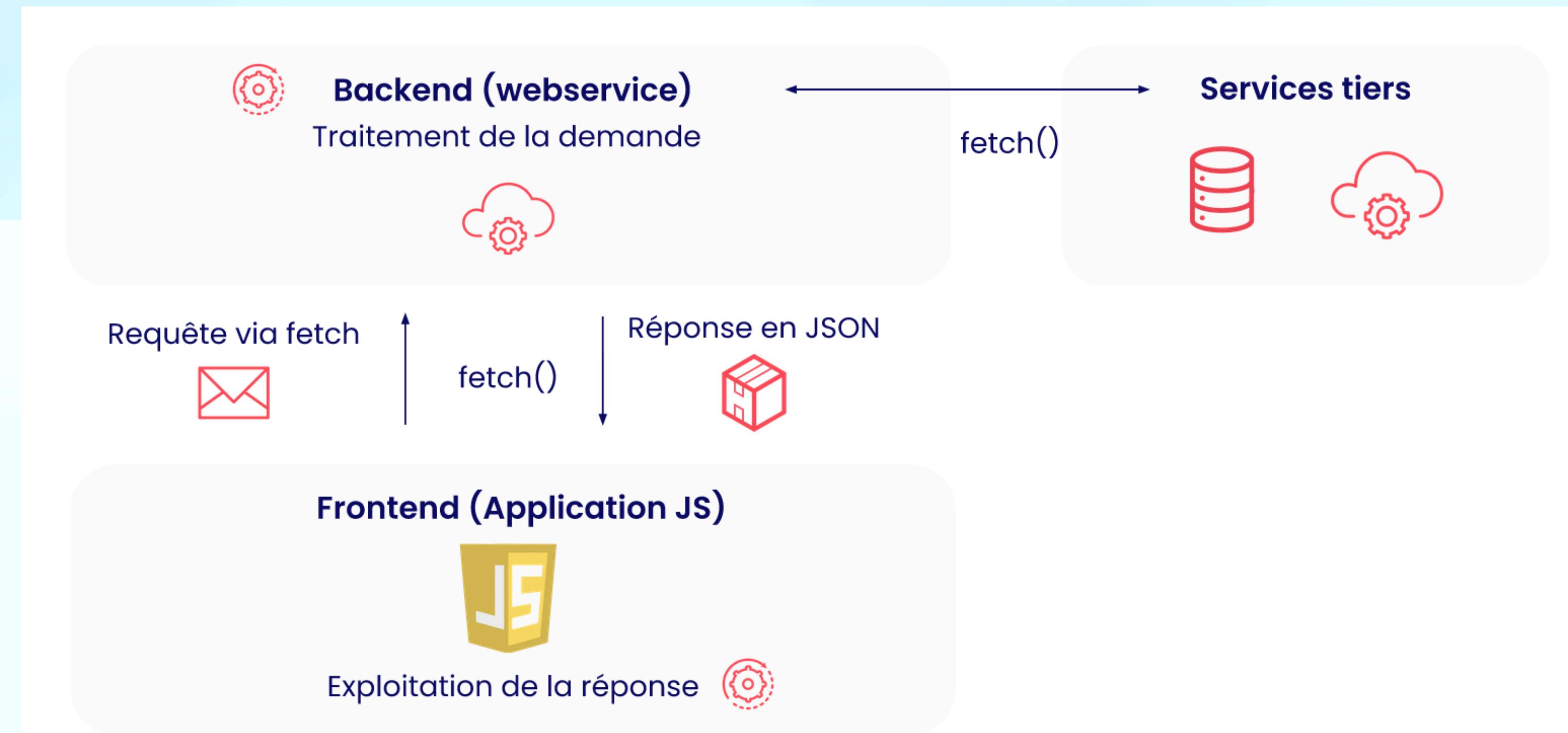
Les services tiers permettent d'enrichir votre webservice pour le rendre plus complet et ainsi avoir une application web plus performante.

Par exemple, une application JS frontend (HTML, CSS & DOM) est incapable de communiquer directement avec MongoDB, elle sera obligée de passer par un backend pour échanger avec la base de données.

Les services tiers

Fonctionnement

Ce schéma illustre les différents échanges entre une application JS, un webservice et des services tiers comme MongoDB ou une API protégée.



Les services tiers

Fonctionnement

Communication avec MongoDB

La mécanique de Mongoose reste inchangée et s'intègre directement dans les routes.

backend/routes/index.js

```
require('../models/connection');
const User = require('../models/users');

router.get('/users', (req, res) => {
  User.find().then(data => {
    res.json({ allUsers: data });
  });
});
```

La route **users** permet de récupérer la liste de tous les utilisateurs de la collection **User**.

Les services tiers

Fonctionnement

Communication avec un webservice tiers

Certaines API sont protégées par une mécanique d'authentification et ne peuvent pas être utilisées côté frontend. Il est alors possible de leur faire des requêtes depuis le backend à l'intérieur d'une route.

backend/routes/index.js

⚠️ Attention : le module `fetch` n'est pas installé par défaut côté backend.

Vous devrez l'installer en tapant **npm install node-fetch@2** à la racine de votre backend.

```
const fetch = require('node-fetch');

router.get('/bitcoin', (req, res) => {
  fetch('https://api.bitcoin.com/price?apiKey=12345')
    .then(response => response.json())
    .then(data => {
      res.json({ bitcoinInfos: data });
    });
});
```

Les services tiers

Fonctionnement

Intégration côté frontend

Côté frontend, il n'y a aucun changement particulier à faire. Votre webservice est joint via un fetch classique à l'adresse `http://localhost:3000`.

frontend/script.js

```
fetch('http://localhost:3000/bitcoin')
  .then(response => response.json())
  .then(data => {
    console.log(data.bitcoinInfos);
});
```

Résultat

```
{
  "name": "BTC",
  "usdPrice": 36679,
  "eurPrice": 38885,
}
```

Les services tiers

Fonctionnement

Fetch une route POST

Par défaut, la méthode fetch envoie une requête de type GET. Il est nécessaire d'ajouter d'autres paramètres pour une requête de type POST.

frontend/script.js

Données à envoyer
disponible dans
req.body côté backend.

```
const data = {  
    firstname: "John",  
    lastname: "Doe"  
}  
  
fetch('http://localhost:3000/save', {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json' },  
    body: JSON.stringify(data)  
})  
.then(response => response.json())  
.then(data => console.log(data))
```

Type de la requête
Mise en forme des
données au format JSON

Informations sur le
format du body

Les services tiers

Fonctionnement

Fetch une route POST

La méthode fetch prend un deuxième paramètre qui se présente sous forme d'objet et contient des options nécessaire à un fetch d'un autre type que GET.

- Le champ **method** contient le type de requête.
- Le champ **headers** permet de préciser dans quel format seront envoyées les informations du body.
- Le champ **body** contient l'objet préparé en amont.

La méthode JSON.stringify() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify) permet de préparer les données car elles ne peuvent pas être envoyées sous forme d'objet mais doivent être converties en chaîne de caractères.

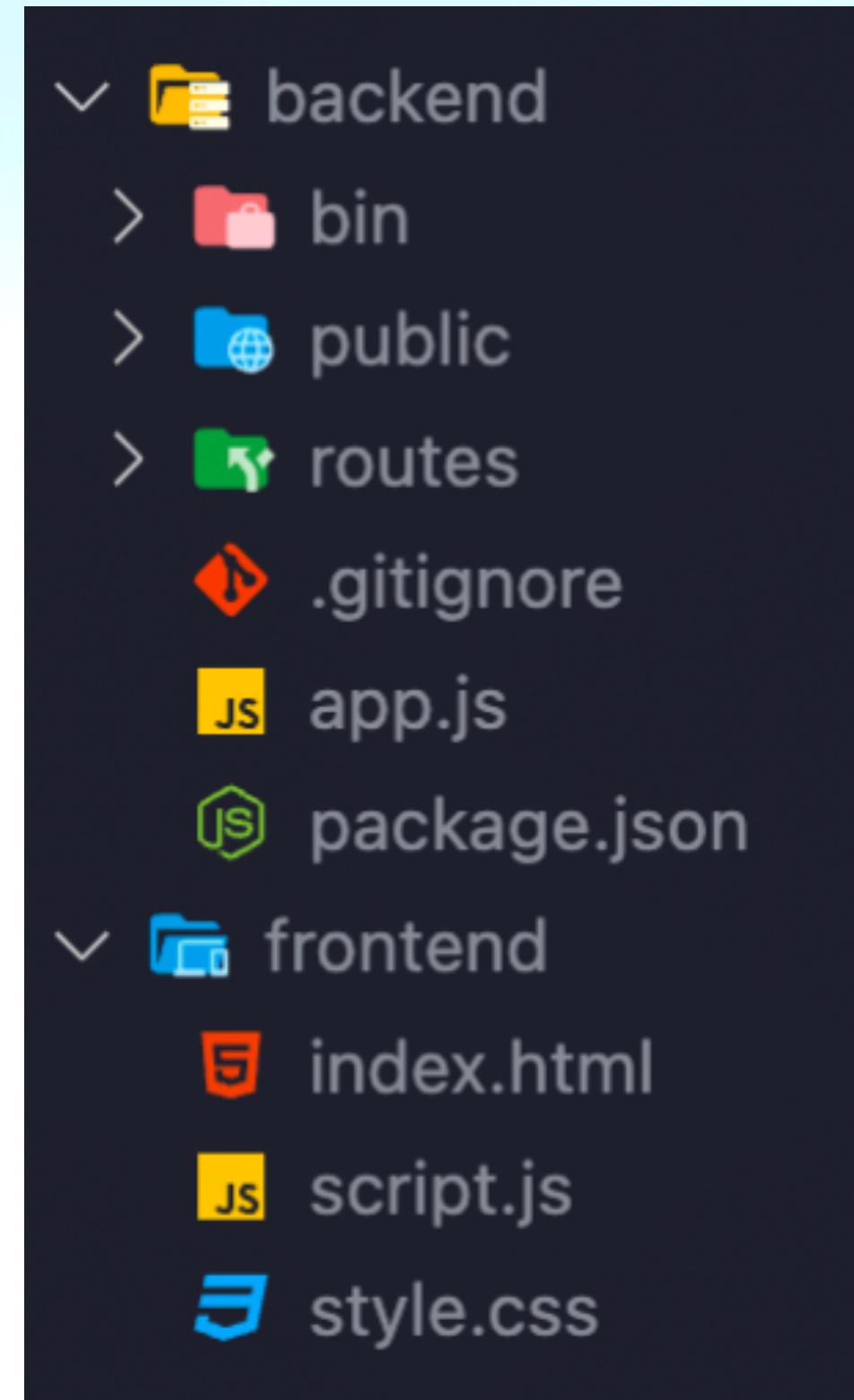
Toutes les informations contenues dans le champ body seront accessibles côté backend dans req.body.

Les services tiers

Fonctionnement

Structure de l'application

L'organisation de votre application est primordiale, créez un répertoire pour le frontend et un répertoire séparé pour le backend.



Challenge #6

WeatherApp - part 2

La seconde partie de ce projet a pour objectif d'intégrer un service tiers au backend webservice : l'API OpenWeatherMap.

Votre application va également devenir plus concrète car elle possède maintenant une partie frontend avec laquelle vous allez devoir communiquer pour afficher la météo des villes choisies.



Challenge #6

WeatherApp - part 2

Environnement de travail

Faites un git clone de <https://github.com/HedzDev/weatherapp-part2.git>

- 👉 Ouvrez deux terminaux afin de récupérer les parties backend et frontend en parallèle.
- 👉 Dans votre premier terminal, positionnez-vous dans le dossier "backend" et exécutez la commande "npm install" pour installer les dépendances.
- 👉 Enfin, toujours dans le backend, installez le module node-fetch avec la commande "npm install node-fetch@2".
- 👉 Importez le module node-fetch dans le fichier routes/index.js.

Challenge #6

WeatherApp - part 2

Mise en place d'un service tiers

👉 Allez sur l'API [Openweathermap](#) et créez vous un compte afin de pouvoir récupérer une clé d'API. Vous pouvez souscrire à la version gratuite de "Current Weather Data".

👉 Lisez la [documentation](#) de l'API pour comprendre comment formuler une requête correctement en fonction des informations souhaitées.

👉 Modifiez maintenant la route POST /weather créée la veille pour qu'elle s'adapte aux données renvoyées par le service tiers. Les routes à modifier sont les suivantes :

- POST /weather : réceptionne la ville présente dans req.body.cityName, l'envoie à Openweathermap qui communique ses informations météorologiques en degrés celsius et enregistre la ville et ses informations dans le tableau "weather".

💡 Dans Openweathermap, les informations en degrés celsius s'obtiennent en ajoutant **&units=metric** à la fin de la requête.

Faites en sorte que votre réponse soit personnalisée en cas d'ajout de doublon et pensez à gérer la casse.

👉 Vérifiez votre route en la testant avec l'extension ThunderClient. Si besoin n'oubliez pas de redémarrer votre projet avec la commande "npm start".

Challenge #6

WeatherApp - part 2

Intégration frontend

Maintenant que la route POST /weather est branchée à l'API OpenWeatherMap côté backend, il est temps de retourner côté frontend afin de les intégrer et rendre votre application dynamique.

Chaque route devra être déclenchée à un moment précis de l'utilisation de l'application. Il faudra que vous mettiez en place la communication entre votre frontend et le backend pour que, dès qu'un événement précis se produise, la bonne route soit déclenchée et que les bonnes informations parviennent.

👉 Intégrez les routes suivantes dans le fichier script.js :

- Dès l'arrivée sur la page de l'application, lancez la requête GET /weather pour afficher les villes déjà enregistrées côté backend.
- A chaque ajout de ville avec l'input dans le header, lancez une requête POST /weather qui ajoutera la ville recherchée et l'affichera directement à la suite des autres sur l'application.
- Au clic sur le bouton “X”, lancez une requête DELETE /weather pour retirer la ville de la liste et de l'affichage.

MVC & Modules



MVC

Contexte

Plus une application va grossir en fonctionnalités, plus elle va avoir de code et il sera difficile de s'y retrouver.

Le principe d'un design pattern est de proposer des solutions d'architecture pour résoudre des problématiques récurrentes. Il est par exemple fréquent de mélanger différentes parties d'une application (frontend, base de données, backend...) ou d'avoir des fichiers de routes tellement chargés qu'ils en deviennent illisibles.

Le design pattern permet d'avoir un code plus "propre" et "rangé" ce qui est essentiel dans une pluralité de contextes comme le travail en équipe ou sur des projets de grande envergure.

Le modèle MVC est le design pattern le plus courant dans l'univers du développement. Il sépare une application en trois composants basiques : le Model, la View et le Controller.

MVC

Model

Le modèle est responsable de la communication avec la base de données et de la description des informations. Dans une architecture, le modèle est matérialisé par le répertoire `models` qui contient :

- La connexion à MongoDB
- Les schémas des collections
- Les modèles des collections

MVC

View

La vue est responsable de l'affichage des informations. Dans une architecture, la vue est matérialisée par le répertoire frontend qui contient :

- Les fichiers HTML & CSS renvoyés au client
- Les fichiers JS côté frontend

MVC

Controller

Le contrôleur est responsable du traitement des requêtes envoyées par le JavaScript côté frontend. Il est le pont entre le Modèle et la Vue. Dans une architecture, le contrôleur est matérialisé par le répertoire routes qui contient :

- La définition des routes
- L'import et l'utilisation des modèles

MVC & Modules

Il existe différentes façons d'implémenter le modèle MVC, en tant que développeur junior il est surtout important pour vous de savoir à quoi correspond un design pattern et quel est son utilité.

Le modèle MVC est également complété par des **modules** qui viennent alléger et optimiser le code en factorisant, par exemple, des fonctions.

Modules

Fonctionnement

L'architecture MVC s'organise directement dans vos répertoires VSCode, les modules quant à eux sont à mettre en place dans votre code.

Pour illustrer le concept de modules, observez l'exemple ci-dessous :

routes/index.js

```
router.post('/add', (req, res) => {
  console.log('admin called POST /add');

  res.json({ result: true });
});

router.put('/update', (req, res) => {
  console.log('admin called PUT /update');

  res.json({ result: true });
});

router.delete('/delete', (req, res) => {
  console.log('admin called DELETE /delete');

  res.json({ result: true });
});
```

Modules

Fonctionnement

Les trois instructions **console.log()** sont redondantes.

Les modules NodeJS permettent de déporter du code dans une boîte hermétique qui pourra être utilisée dès que nécessaire.

Modules

Fonctionnement

Préparation du module

La première étape de l'utilisation d'un module est de séparer le code qu'on souhaite répéter dans un autre fichier sous la forme d'une fonction.

routes/index.js

```
router.post('/add', (req, res) => {
  console.log('admin called POST /add');

  res.json({ result: true });
});

router.put('/update', (req, res) => {
  console.log('admin called PUT /update');

  res.json({ result: true });
});
```

modules/displayAdmin.js

```
function displayAdmin(route) {
  console.log(`admin called ${route}`);
}
```

Modules

Fonctionnement

Export du module

La seconde étape consiste à s'assurer que chaque élément dont vous aurez besoin soit explicitement exporté.

modules/displayAdmin.js

```
function displayAdmin(route) {  
  console.log(`admin called ${route}`);  
}  
  
module.exports = { displayAdmin };
```

Modules

Fonctionnement

Import du module

Enfin, la dernière étape est d'importer chaque élément exporté afin de l'utiliser dans votre code.

routes/index.js

```
const { displayAdmin } = require('../modules/displayAdmin');

router.post('/add', (req, res) => {
    displayAdmin('POST /add');
    res.json({ result: true });
});

router.put('/update', (req, res) => {
    displayAdmin('PUT /update');
    res.json({ result: true });
});
```

MVC & Modules

Organisation des routes

En s'inspirant de la logique des modules, vous pouvez également séparer et regrouper vos routes afin de les rendre plus lisibles et maintenables.

MVC & Modules

Organisation des routes

Regroupement par famille

Express vous permet de séparer vos routes en différentes "familles" où chacune d'elle aura son propre préfixe.

Une famille = un fichier js.

famille /

/home

famille /articles

/articles
/articles/new
/articles/promo

MVC & Modules

Organisation des routes

Création d'un préfixe

Le préfixe est le nom choisi pour désigner une famille de routes (par exemple: articles). Celui-ci doit figurer dans le nom du fichier des routes correspondantes (articles.js) mais il n'apparaîtra pas dans les noms donnés à ses routes.

/ équivaut à /articles

/new équivaut à /articles/new

/promo équivaut à /articles/promo

MVC & Modules

Organisation des routes

Création d'un préfixe

routes/articles.js

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

/articles → router.get('/', (req, res) => {
    res.json({ message: 'All articles' });
});

/articles/new → router.get('/new', (req, res) => {
    res.json({ message: 'New articles' });
});

/articles/promo → router.get('/promo', (req, res) => {
    res.json({ message: 'Promo articles' });
});

module.exports = router;
```

MVC & Modules

Organisation des routes

Enregistrement d'un nouveau préfixe

app.js

```
...
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var articlesRouter = require('./routes/articles');

var app = express();

const cors = require('cors');
app.use(cors());

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/articles', articlesRouter);
module.exports = app;
```

Import du
fichier de routes

Utilisation du
nouveau
préfixe /articles

MVC & Modules

Organisation des routes

Tester les différentes routes

Les routes créées dans le fichier index.js sont toujours accessibles de la même manière.

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** http://localhost:3000/home
- Status:** 200 OK
- Size:** 30 Bytes
- Time:** 5 ms

The interface includes tabs for Query, Headers, Auth, Body, and Tests. The Response tab displays the JSON data received from the server:

```
1 {  
2   "message": "Welcome to /home"  
3 }
```

MVC & Modules

Organisation des routes

Tester les différentes routes

Les nouvelles routes de la famille /articles devront être appelées en ajoutant le préfixe.

The screenshot shows a REST client interface with two main sections: a left panel for sending requests and a right panel for viewing responses.

Left Panel (Request):

- Method: GET
- URL: http://localhost:3000/articles/new
- Buttons: Send, Query, Headers 2, Auth, Body, Tests

Right Panel (Response):

- Status: 200 OK, Size: 26 Bytes, Time: 4 ms
- Headers: Response, Headers 7, Cookies, Results, Docs New
- Message content:
 - 1 ↴ {
 - 2 "message": "New articles"
 - 3 }

Challenge #7

WeatherApp - part 3

Côté backend, la troisième partie de ce projet a pour objectif de mieux structurer notre application en organisant nos routes et d'introduire une base de données afin de rendre nos informations pérennes.



Challenge #7

WeatherApp - part 3

- 👉 Commencez par "ranger" les routes weather déjà existantes au sein d'un nouveau fichier **routes/weather.js** qui devra être enregistré auprès du backend avec le préfixe **/weather**.
- 👉 Ajoutez un dossier **models** avec un fichier **cities.js** dans lequel vous créez une collection **cities**. Celle-ci sera par la suite chargée d'enregistrer vos villes favorites et les informations suivantes : cityName (String), main (String), description (String), tempMin et tempMax (Number).
- 👉 Importez votre model dans le fichier routes/weather.js.
- 👉 Toujours dans le dossier models, ajoutez le fichier de connexion à votre base de données.
- 👉 Importez votre fichier de connexion dans le fichier app.js.

Challenge #7

WeatherApp - part 3

- 👉 Votre collection cities remplace maintenant l'ancienne variable globale "weather" que vous pouvez supprimer.
- 👉 Modifiez vos routes pour qu'elles s'adaptent à votre base de données. Les routes à modifier sont les suivantes :
 - POST /weather : réceptionne la ville présente dans req.body.cityName, l'envoie à Openweathermap qui communique ses informations météorologiques en degrés celsius et enregistre la ville et ses informations en BDD.

Faites en sorte que votre réponse soit personnalisée en cas d'ajout de doublons dans la BDD et pensez à gérer la casse.

- 💡 La casse peut directement être gérée avec Mongoose et le constructeur RegExp (la gestion de la casse avec RegExp se fait avec le paramètre « i »).

Challenge #7

WeatherApp - part 3

- GET /weather : renvoie la liste de toutes les villes enregistrées en BDD.
- GET /weather/:cityName : renvoie uniquement les informations présentes en BDD de la ville ciblée.

Gérer votre réponse de requête dans le cas où la ville ciblée n'existe pas en BDD et pensez à gérer la casse.

- DELETE /weather/:cityName : supprime la ville ciblée de la collection cities en BDD et renvoie la liste des villes restantes.

Gérer votre réponse de requête dans le cas où la ville ciblée n'existe pas en BDD et pensez à gérer la casse.

👉 Vérifiez vos routes en les testant avec l'extension ThunderClient. Si besoin n'oubliez pas de redémarrer votre projet avec la commande **npm start**.

Authentification



Authentification

Contexte

Lors de tous développements, le développeur doit respecter des bonnes pratiques visant à protéger les applications contre d'éventuelles menaces. Les bonnes pratiques que nous aborderons aujourd'hui sont :



Sécuriser les échanges lors de l'authentification



Restreindre l'accès à une ressource



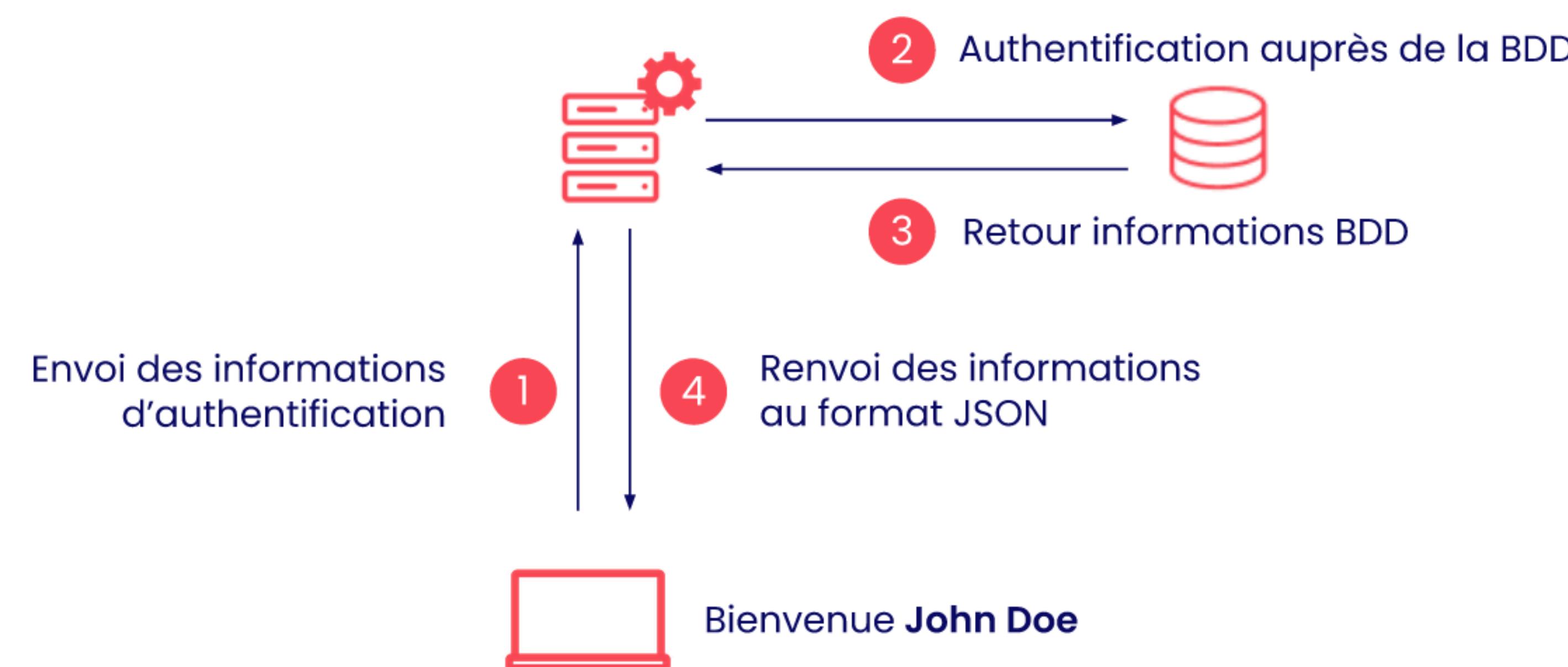
Ne pas stocker des mots de passe en clair

Authentification

Fonctionnement

Première requête avec le backend

Le premier échange entre le frontend et le backend a lieu pour une mécanique d'authentification (inscription ou connexion).

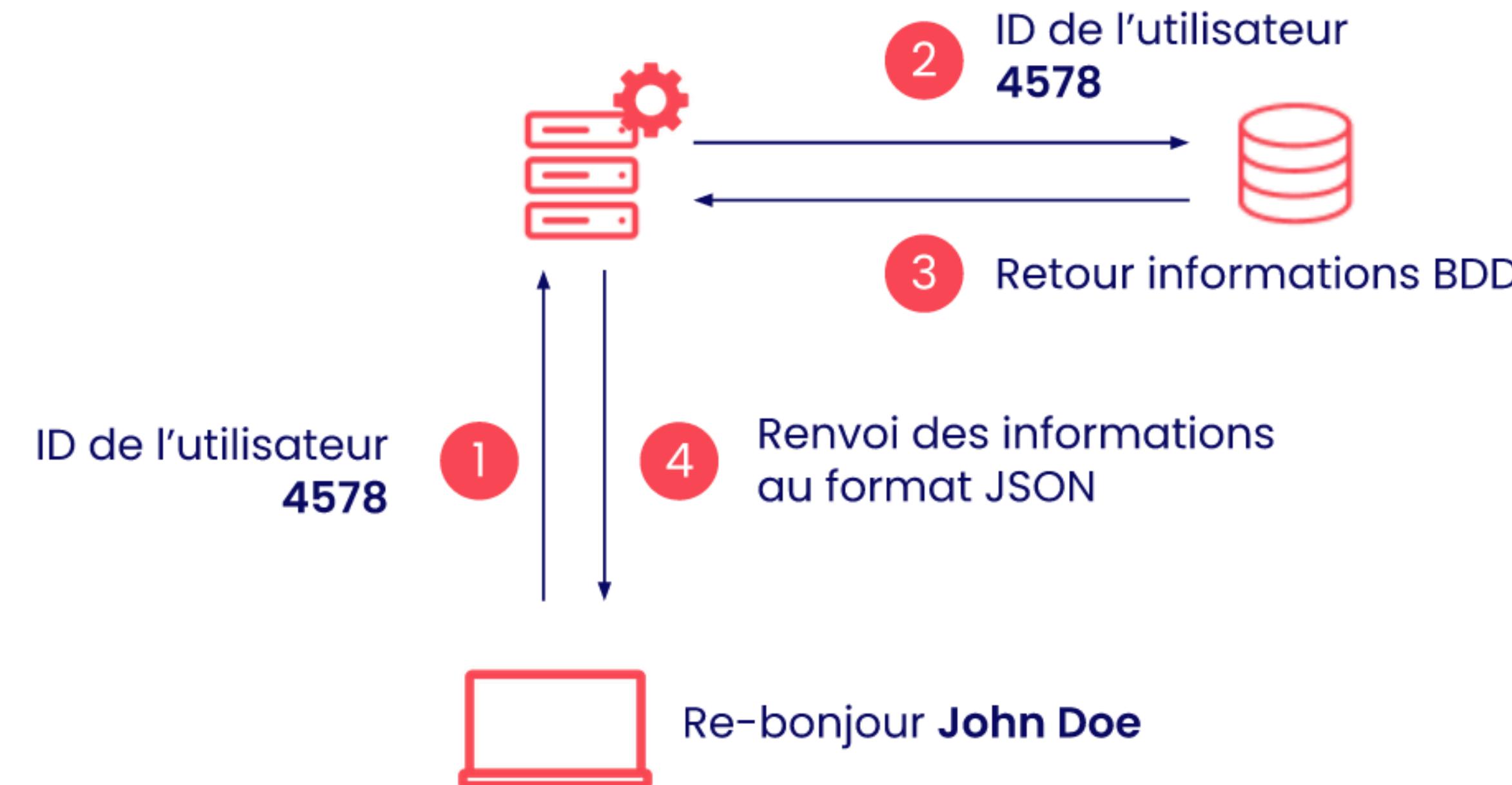


Authentification

Fonctionnement

Resolliciter la base de données

Chaque nouvelle requête du frontend devra fournir l'ID de l'utilisateur pour que le backend puisse interroger la base de données.



Authentification

Fonctionnement

Sécuriser les échanges

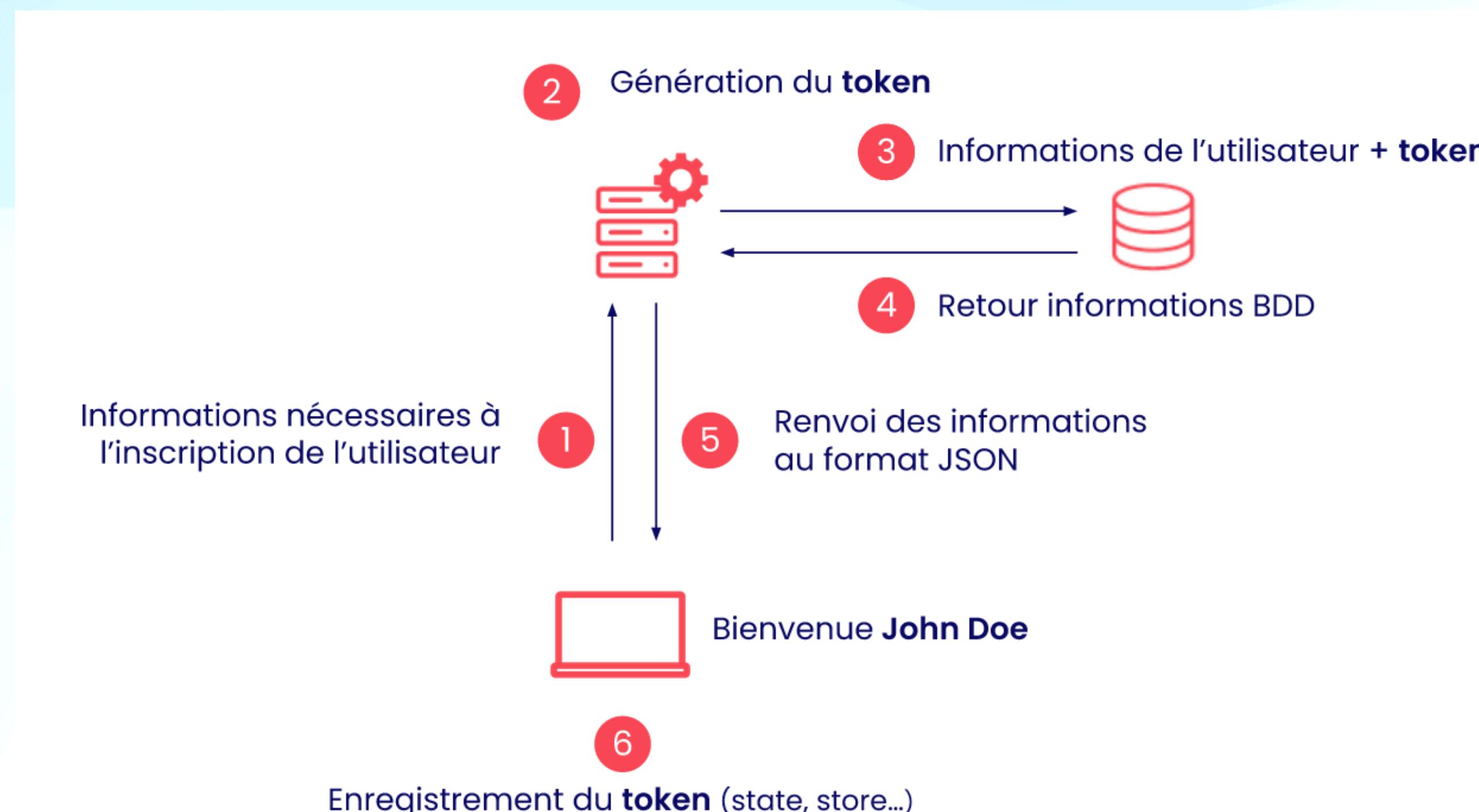
Problème : Chacun de ces échanges représente une faille de sécurité possible.
Même si l'ID est unique, il peut fuité ou être usurpé sans possibilité de le re-générer 😞

Authentification

Fonctionnement

Sécuriser les échanges

Solution : La solution est de sécuriser l'échange en générant et renvoyant un token au frontend au moment de la création du compte de l'utilisateur.



Authentification

Mécanique du token

Le principe

Lors de son authentification, on associe à l'utilisateur un identifiant qui permettra de vérifier son identité tout au long de la navigation.

Le token va être généré côté backend dans la route qui récupère les informations à sécuriser.

Authentification

Mécanique du token

Le module uid2

Le module **uid2** est chargé de générer une chaîne de caractères de 32 caractères aléatoires et uniques en guise de token.



npm install uid2

Génération du token

```
const uid2 = require('uid2');
...
const token = uid2(32);
```

Authentification

Mécanique du token

Impact sur la BDD

Le schéma chargé de définir la collection des utilisateurs devra être mis à jour en conséquence.

backend/models/users.js

```
const userSchema = mongoose.Schema({  
    username: String,  
    password: String,  
    token: String,  
});
```

Authentification

Mécanique de hachage

Principe

Le hachage permet d'obfuscuer un mot de passe à l'aide d'une fonction cryptographique irréversible.

Le nombre d'itérations de hachage correspond au “coût”, c'est-à-dire le nombre de fois où le mot de passe va passer par la fonction de hachage : c'est un compromis entre sécurité et rapidité (plus on hash, plus c'est long).

Authentification

Mécanique de hachage

Principe

Mot de passe en clair
azerty123

Nombre d'itérations de hachage
10



Fonction de hachage

Mot de passe haché
LQOGRdFTLqAtrJgHz5arB5AC...

Authentification

Mécanique de hachage

Le module bcrypt

Le module **bcrypt** sera en charge du hachage du mot de passe et de la vérification de ce dernier.



npm install bcrypt

Hachage via la méthode hashSync

```
const bcrypt = require('bcrypt');
...
const hash = bcrypt.hashSync('password', 10);
```

Authentification

Mécanique de hachage

En pratique

Le "hash" sera enregistré à la place du mot de passe en clair.

backend/routes/users.js

```
const hash = bcrypt.hashSync(req.body.password, 10);

const newUser = new User({
  username: req.body.username,
  password: hash,
  token: uid2(32),
});
```

Authentification

Mécanique de hachage

Connexion

Lors de la connexion, le mot de passe reçu sera comparé avec celui enregistré en base de données via la méthode compareSync.

backend/routes/users.js

```
User.findOne({ username: req.body.username }).then(data => {
  if (bcrypt.compareSync(req.body.password, data.password)) {
    res.json({ result: true });
  } else {
    res.json({ result: false });
  }
});
```

Challenge #8

WeatherApp - part 4

Dans la dernière partie de cette app nous allons mettre en place la possibilité de créer un compte et de se connecter.

Nous allons aussi créer un module qui permettra de vérifier si tous les champs inputs ont bien été entrer.



Challenge #8

WeatherApp - part 4

- 👉 Faites un git clone de <https://github.com/HedzDev/weatherapp-part4.git>
- 👉 Faites un npm install dans le dossier backend et dans le dossier frontend pour installer les dépendances.

Côté Backend

- 👉 Dans le dossier **models**, créez un fichier users.js dans lequel il faudra mettre en place le schéma et le model de la collection users. Un document **user** devra contenir les champs name (string), email (string) et password (string).
⚠ Pour créer un compte, un user devra obligatoirement entrer les 3 champs name, email et password. Pour se connecter il ne devra entrer que les champs email et password.
- 👉 Dans le dossier routes, créez un fichier user.js dans lequel faudra implémenter une route /signup et une route /signin.

Faites en sorte que votre réponse soit personnalisée en cas d'ajout de doublons d'email dans la BDD ou bien en cas d'user inexistant au niveau de la connexion.

- 👉 Créez un dossier modules, il faudra ajouter dans celui-ci un fichier nommé checkBody.js qui devra contenir une fonction qui viendra vérifier que tous les champs inputs sont bien entrés.
- 👉 Implémentez cette fonction dans les routes /signup et /signin afin de bloquer l'exécution du code si tous les champs ne sont pas bien renseignés.

Challenge #8

WeatherApp - part 4

Côté Frontend

- 👉 Dans le fichier login.js implémentez la logique nécessaire pour envoyer les données des formulaires de login côté backend.
 - 💡 Si la data renvoyée par le backend est correcte, utilisez **window.location.assign('index.html')** afin d'envoyer l'utilisateur sur la page index.html une fois l'utilisateur bien authentifié.