

Support de cours

Formation Full Stack MERN



Formateur : Kamel
ABBASSI

Introduction

1. Présentation de l'application finale

Contacts Manager V1.0

La liste des contacts

#	Nom	Tél	Actions
1	kamel abbassi	87686760	<button>Modifier</button> <button>supprimer</button>
2	maryem hdaouadi	23456789	<button>Modifier</button> <button>supprimer</button>
3	sonia marouk	29456780	<button>Modifier</button> <button>supprimer</button>

Ajouter un contact

Nom:

Tél:

Ajouter

Modifier contact

ID: 65513cb9cd991ecdca93ee0c

Nom:

Tél:

Close Save Changes

Supprimer contact

Voulez vraiment supprimer ce contact ?

ID: 65513cb9cd991ecdca93ee0c

Close Save Changes

MongoDB Compass - localhost:27017/merndb.contacts

localhost:27017

Documents merndb.contacts

My Queries

Databases

admin

config

local

merndb

contacts

merndb.contacts

1 DOCUMENTS 2 INDEXES

Documents Aggregations Schema Indexes Validation

Filter Type a query: { field: 'value' } or Generate query Explain Reset Find Options

ADD DATA EXPORT DATA

1 - 5 of 5

#	_id	Objectid	nom String	tel String	createdat Date
1	Objectid("6550f681cd991ecdca8...)	"paul"	"87686768"	2023-11-12T16:00:08	
2	Objectid("6550f69cd991ecdca8...)	"hh"	"44444444"	2023-11-12T16:00:21	
3	Objectid("6550f6a4cd991ecdca8...)	"hh"	"44444443"	2023-11-12T16:00:31	
4	Objectid("6550f6c8cd991ecdca8...)	"valid"	"56789043"	2023-11-12T16:25:41	
5	Objectid("65513cb9cd991ecdca9...)	"kamel abbassi"	"87686760"	2023-11-12T20:59:13	

http://localhost:3000/contact/ajouter

POST http://localhost:3000/contact/ajouter

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary

Key	Value	Bulk Edit
<input checked="" type="checkbox"/> nom	samir guesmi	
<input checked="" type="checkbox"/> tel	21626359	
Key	Value	

400 Bad Request 47 ms 334 B Save Response

1

2. Ce que vous devriez savoir

- Utiliser HTML et CSS
- Programmer avec JavaScript
- Développer des projets Node.js
- Installer et utiliser des dépendances
 - Gestionnaire de paquets : npm ou yarn
- Avoir une petite expérience avec reactJS

Le projet et les technologies de développement

Back-end : Projet Node et Express

- Dernière version de Node.js
- Express.js
- MongoDB
- Babel
- Nodemon
- Cors

Front-end : Projet React

- React CLI
- Bootstrap v5.1
- Axios

Déploiement, hébergement et autres

- Git
- Heroku
- Compass
- Postman
- React Developer Tools

Projet full-stack avec MERN

- MongoDB
- Express.js
- React.js
- Node.js

Environnement de travail

1. Editeur de code :

- Permet de gérer les fichiers de votre projet, Ajouter des extensions pour optimiser le développement.



<https://code.visualstudio.com/>

2. NodeJS

Ce logiciel permet d'exécuter code js coté serveur sans besoin d'un navigateur

Dans ce logiciel on a les deux gestionnaires de paquets npm et yarn



<https://nodejs.org/fr>

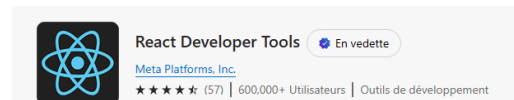
3. React CLI

C'est l'interface de ligne de commande React CLI pour lancer des commandes qui permet de créer des projets react dans une seule ligne.

4. React Developer Tools

C'est une extension de navigateur web pour analyser, visualiser l'arborescence de votre composant React et déboguer le fonctionnement de l'application.

<https://react.dev/learn/react-developer-tools>



5. SGBD MongoDB

<https://www.mongodb.com/fr-fr>

<https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/>

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



6. Postman

Postman est un logiciel très populaire qui permet aux développeurs de tester, de documenter et de gérer les API (interfaces de programmation d'application).



7. Git

Git permet de suivre et de gérer les différentes versions d'un projet logiciel. Il conserve un historique complet des modifications apportées aux fichiers, ce qui facilite la collaboration et le suivi des évolutions du code.

<https://git-scm.com/download/win>

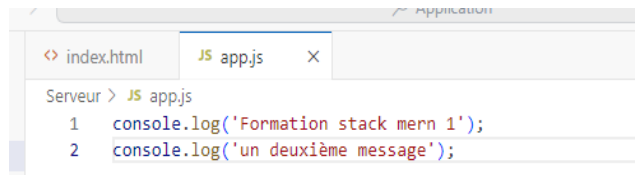
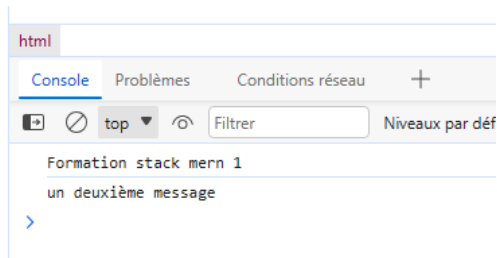
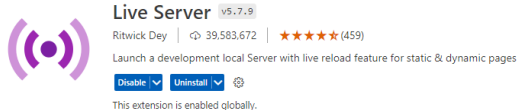


Application serveur : JavaScripts

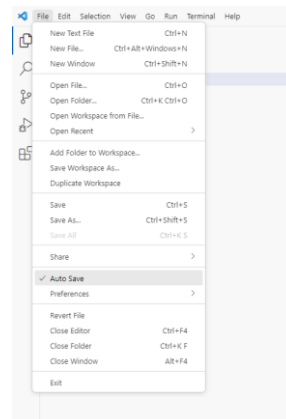
3. Exercice 1 : Utilisation de console avec JS

```
console.log('Message en JS')
```

4. Ajouter l'extension « live server » pour Visual code



Paramètre pour activer l'enregistrement automatique



Les commentaires



Les types des données en Javascript

JavaScript has 8 Datatypes

String

- ✓ Number
- ✓ BigInt
- ✓ Boolean
- ✓ Undefined

- ✓ Null
- ✓ Symbol
- ✓ Object

Object

The object data type can contain:

- ✓ An object
- ✓ An array
- ✓ A date

Exemples

```
// Numbers:
let length = 16;
let weight = 7.5;

//BigInt
let x =
BigInt("123456789012345678901234567890
");

// Strings:
let color = "Yellow";
let lastName = "Johnson";

// Booleans
let x = true;
let y = false;
```

```
// Object:
const person = {firstName:"John",
lastName:"Doe"};

// Array object:
const cars = ["Saab", "Volvo", "BMW"];

// Date object:
const date = new Date("2022-03-25");

let x;           // Now x is undefined
x = 5;           // Now x is a Number
x = "John";      // Now x is a String
```

```
a = 2; // assignment
// number ( + , - , * , / , % , ** )
var b = 3;

var somme = a + b ;
var sous = a - b;
var multi = a * b ;
var div = a/b;

var rest = a % b;
var p = a ** b;

console.log(somme);
console.log(sous);
console.log(multi);
```

```
console.log(div);

console.log(rest);

console.log(p);
```

5. Exercices sur les tableaux

6. Exercices sur les objets

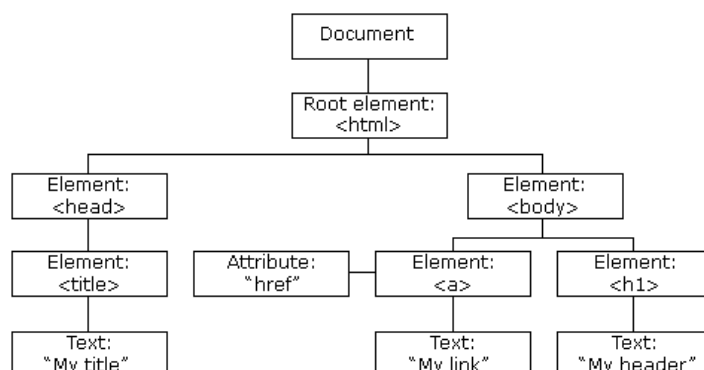
```
var personne ={
  "nom": "ABBASSI",
  "tel": 45564654,
  "email": "abbassi.kamel@gmail.com",
  "skills": ["js", "html", "nodejs", "php"]
}
console.log(personne)
```

```
▼ {nom: 'ABBASSI', tel: 45564654, email: 'abbassi.kamel@gmail.com', skils: Array(4)} ⓘ
  email: "abbassi.kamel@gmail.com"
  nom: "ABBASSI"
  ► skils: (4) ['js', 'html', 'nodejs', 'php']
  tel: 45564654
  ► [[Prototype]]: Object
```

7. Exercices sur les fonctions

8. Exercices sur le DOM

The HTML DOM Tree of Objects



L'arbre des objets du DOM HTML est une représentation hiérarchique des éléments HTML d'une page web. Organisés comme un arbre, ces objets structurés permettent au navigateur de comprendre, manipuler et afficher le contenu de la page. Chaque élément, tel que les balises, les attributs et leur contenu, est représenté par un nœud dans cet arbre. Cette structure organise les relations parent-enfant entre les éléments, facilitant la navigation et la modification du contenu par le biais du JavaScript. Cela permet aux développeurs web de dynamiser et de modifier la page en utilisant des langages comme JavaScript, CSS ou HTML.

```
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

https://www.w3schools.com/js/js_htmlDOM_document.asp

```
<html>
<body>

<p id="p2">Hello World!</p>

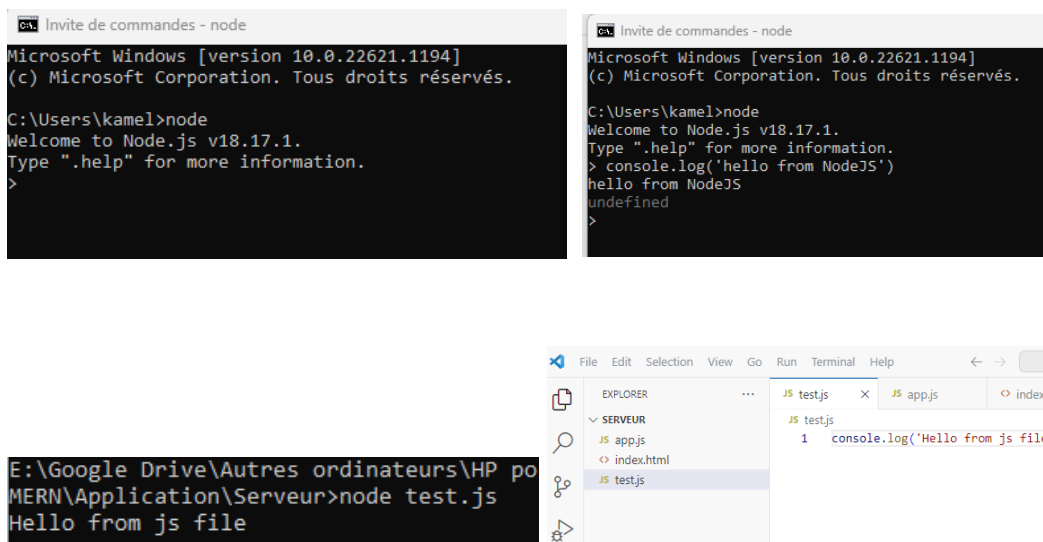
<script>
document.getElementById("p2").style.color = "blue";
</script>

</body>
</html>
```


Application serveur : NodeJS

Node.js est un environnement d'exécution JavaScript côté serveur, basé sur le moteur V8 de Google Chrome. Il permet d'exécuter du code JavaScript hors d'un navigateur, ouvrant ainsi la voie à des applications côté serveur rapides et évolutives. Node.js est réputé pour sa gestion asynchrone et non bloquante, idéale pour des applications gérant de multiples opérations simultanées. Il offre un écosystème riche de modules (via npm) facilitant le développement d'applications web, les API, les serveurs et les outils en JavaScript, offrant ainsi aux développeurs une plateforme puissante pour créer des solutions performantes et évolutives.

NodeJS : Environnement pour exécuter javascript en dehors de navigateur



Installer npm (node package manager) ! gestion de package dans nodejs

Créer le projet Node

Package.json

npm init

Point d'entrée: index.js

Lancer le projet

node index.js

9. Créer un serveur avec nodejs

1. Lancer la commande **npm init** dans le dossier serveur pour initialiser le projet
2. Reprendre aux questions comme suite :

```

1  {
2    "name": "serveur",
3    "version": "1.0.0",
4    "description": "application serveur (backEnd)",
5    "main": "app.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [
10     "formation",
11     "mern",
12     "crud",
13     "api"
14   ],
15   "author": "Kamel ABBASSI",
16   "license": "ISC"
17 }
18

```

10. Ajouter un fichier index avec le code suivant

```
console.log('Hello Word');
```

11. Exécuter l'application

```
node index.js
```

```
PS C:\Users\kamel\Desktop\Application V2> node index.js
Hello Word
```

3. Installer le package Expressjs

<https://expressjs.com/fr>

```
npm install express --save
```

12. Tour d'horizon sur les applications d'expressjs (site officiel)

Pour comprendre les paramètres avec les méthodes GET et POST, vous pouvez créer des exercices simples. Voici quelques exemples d'exercices que vous pouvez utiliser pour vous familiariser avec ces concepts :

Exercice 1 : Paramètres GET

Créez une application Express qui accepte une requête GET à l'URL `/hello` avec un paramètre `name`. L'application devrait renvoyer un message de salutation en utilisant le nom fourni.

```

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

app.get('/hello', (req, res) => {
  const name = req.query.name;
  if (name) {
    res.send(`Hello, ${name}!`);
  } else {
    res.send('Hello, World!');
  }
});

app.listen(3000, () => {
  console.log('Serveur Express en cours d\'exécution sur le port 3000');
});

```

Pour tester cet exemple :

<http://localhost:3000/hello?name=abbassi>

Exercice 2 : Paramètres POST

Créez une application Express qui accepte une requête POST à l'URL `"/add"` avec deux paramètres `"num1"` et `"num2"`. L'application devrait renvoyer la somme des deux nombres.

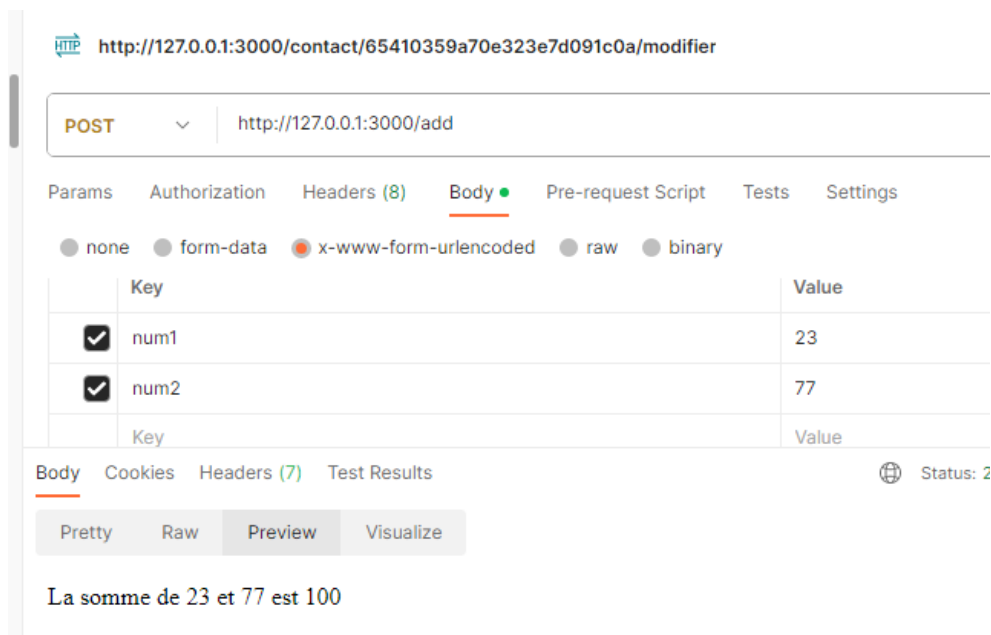
```
const express = require('express'); // Importez le module Express
const bodyParser = require('body-parser'); // Importez le module body-parser pour gérer les données POST
const app = express(); // Créez une instance de l'application Express

app.use(bodyParser.urlencoded({ extended: true })); // Utilisez body-parser pour analyser les données POST

// Définissez une route pour gérer les requêtes POST à l'URL "/add"
app.post('/add', (req, res) => {
  const num1 = parseInt(req.body.num1); // Récupérez le premier paramètre "num1" de la requête POST
  const num2 = parseInt(req.body.num2); // Récupérez le deuxième paramètre "num2" de la requête POST

  if (!isNaN(num1) && !isNaN(num2)) { // Vérifiez si les paramètres sont des nombres valides
    const sum = num1 + num2; // Calculez la somme des deux nombres
    res.send(`La somme de ${num1} et ${num2} est ${sum}`); // Renvoyez la réponse avec la somme
  } else {
    res.status(400).send('Les paramètres num1 et num2 doivent être des nombres valides.');// Si les paramètres ne sont pas valides, renvoyez une erreur 400
  }
});

app.listen(3000, () => {
  console.log('Serveur Express en cours d\'exécution sur le port 3000'); // Écoutez les requêtes sur le port 3000
});
```



Pour ces exercices, vous pouvez utiliser Postman, Curl, ou un navigateur pour tester les requêtes GET et POST avec différents paramètres. Cela vous permettra de comprendre comment Express gère les paramètres de requête et les paramètres POST dans vos applications web.

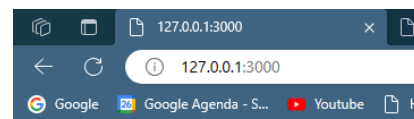
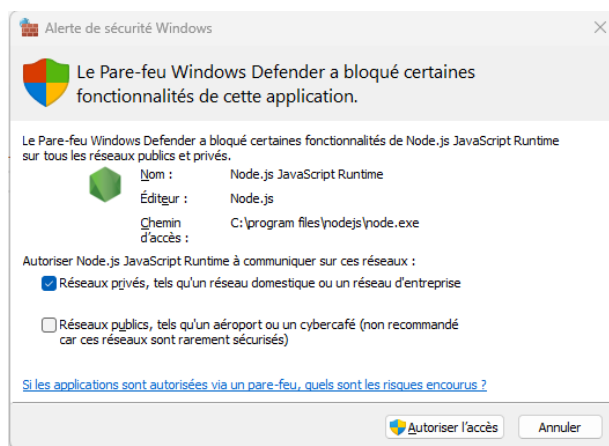
13. Effacer le contenu du fichier index.js

14. Créer notre premier serveur

```
const express = require('express')
const bodyParser = require('body-parser');
const app = express()
const port = 3000
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

15. Lancer le serveur

```
node app.js
```



Server worked !

16. Ajouter les routes suivantes

```
app.get('/home', (req, res) => {
  res.send('Bienvenue dans la page home')
})
```

17. Exercice :

1. Déclarer 3 étudiants sous forme des objets JS, et chaque étudiant possède nom, prénom et un tableau de 3 notes (Math, Physique et anglais)
2. Créer une méthode **getFullName()** qui permet d'afficher le nom et prénom d'un étudiant
3. Définir la méthode **moyenneGenarle()**, qui permet de calculer la moyenne générale d'un étudiant donné
4. Ecrire un tableau étudiants qui regroupe tous les étudiants en utilisant la méthode push ().
5. Ecrire une méthode **getFirstStudent ()** pour déterminer le meilleur étudiant

```
etudiant1 = {
  nom: 'ABBASSI',
  prenom: 'KAMEL',
  notes: {
    'math': 12.5,
    'physique': 13,
    'anglais': 16
  },
  moyG: null
}
etudiant2 = {
  nom: 'AMRI',
  prenom: 'SOFIEN',
  notes: {
    'math': 17.5,
    'physique': 19,
    'anglais': 12
  },
  moyG: null
}
etudiant3 = {
  nom: 'HADDAD',
  prenom: 'FAYCEL',
  notes: {
    'math': 11.5,
    'physique': 11,
    'anglais': 10
  },
  moyG: null
}

function getFullName(e) {
  console.log(e.nom + ' ' + e.prenom)
}

function moyenneGenerale(e) {
  somme = 0;
  for (mat in e.notes) {
    somme = somme + e.notes[mat];
    console.log(somme)
  };
  e.moyG = somme / 3;
  console.log("MG =" + e.moyG);
}

getFullName(etudiant2)

tab = [];
tab.push(etudiant1, etudiant2, etudiant3)

for (let e of tab) {
  moyenneGenerale(e);
}

// La boucle a été exécutée, maintenant nous pouvons afficher tab.
console.log(tab);
```

Ce code JavaScript crée des objets représentant des étudiants avec leurs noms, prénoms et notes dans différentes matières. Ensuite, il définit deux fonctions : `getFullName` pour afficher le nom complet de l'étudiant et `moyenneGenerale` pour calculer la moyenne générale de l'étudiant en se basant sur ses notes. Enfin, il remplit un tableau avec les objets étudiants créés et calcule la moyenne générale pour chacun d'eux.

Voici une explication détaillée du code :

Trois objets sont créés pour représenter trois étudiants avec leurs noms, prénoms et notes dans trois matières (mathématiques, physique et anglais). Chaque objet a une propriété moyG initialisée à null pour stocker la moyenne générale.

Deux fonctions sont définies :

`getFullName(e)` : Cette fonction prend un objet `e` représentant un étudiant en paramètre et affiche son nom complet en combinant les propriétés `nom` et `prenom`.

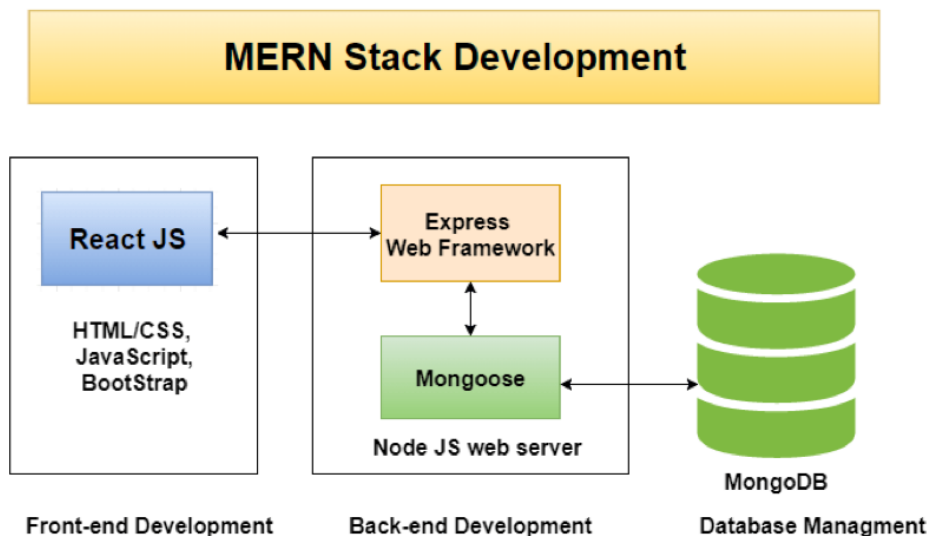
`moyenneGenerale(e)` : Cette fonction prend un objet `e` représentant un étudiant en paramètre. Elle itère à travers les notes de l'étudiant, calcule la somme des notes, puis calcule la moyenne en divisant cette somme par le nombre de matières (dans ce cas, 3). Enfin, elle stocke la moyenne générale dans la propriété `moyG` de l'objet `e`.

Les objets étudiants sont ajoutés à un tableau `tab`.

Une boucle `for...of` parcourt chaque objet étudiant dans le tableau `tab` et appelle la fonction `moyenneGenerale` pour calculer la moyenne générale de chaque étudiant.

Une fois que toutes les moyennes ont été calculées, le contenu du tableau `tab`, qui contient désormais les objets étudiants avec leurs moyennes générales mises à jour, est affiché dans la console.

Application serveur : Architecture MERN (MongoDB, ExpressJS, ReactJS, NodeJS)



18. Installer nodemon

```
npm install nodemon --save-dev  
ou bien  
npm install -g nodemon
```

Ces deux commandes `npm install` permettent d'installer le package Nodemon, un outil très utile pour les développeurs Node.js.

1. `npm install nodemon --save-dev` : Cette commande installe Nodemon localement dans un projet Node.js spécifique. L'option `--save-dev` indique à npm d'ajouter Nodemon en tant que dépendance de développement (`devDependencies`) dans le fichier `package.json` du projet. Les dépendances de développement sont des packages utilisés pour le développement mais pas pour l'exécution en production. L'installation locale signifie que Nodemon ne sera disponible que dans ce projet spécifique.

2. `npm install -g nodemon` : Cette commande installe Nodemon globalement sur votre système. L'option `-g` (ou `--global`) indique à npm d'installer le package de manière globale, ce qui signifie qu'il sera accessible de n'importe où sur votre système. Cela permet d'utiliser Nodemon pour surveiller et redémarrer automatiquement des applications Node.js à partir de n'importe quel répertoire.

L'utilisation de Nodemon simplifie le processus de développement en permettant un redémarrage automatique de votre serveur Node.js à chaque modification de fichier, ce qui est très pratique pour le développement en temps réel. Cette fonctionnalité permet d'éviter de devoir arrêter et relancer manuellement le serveur à chaque modification, ce qui peut être fastidieux et réduit la productivité.

19. Relancer l'application avec la nouvelle package nodemon

```
nodemon app.js
```

```

C:\Windows\system32\cmd.exe - "node" "C:\Users\kamel\AppData\Roaming\npm\
Microsoft Windows [version 10.0.22621.1194]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\kamel\Desktop\Application\Serveur>nodemon app.js
[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
Example app listening on port 3000

```

The screenshot shows a web application with a list of contacts on the left and a form to add a new contact on the right. The list contains three entries, each with a name, phone number, and status indicators. The form has input fields for 'Nom' and 'Tel', and a button labeled 'Ajouter'.

Express
mongoose

```

[
  {
    "nom": "Molecule Man",
    "tel": 29,
  },
  {
    "nom": "Madame Uppercut",
    "tel": 39,
  }
]

```

20. Ajouter les routes nécessaires

```

// API Contacts
//Ajouter contact
app.get('/contact/lister', (req,res)=>{
  //Les traitements necessaires pour lister les contacts
  res.send('Les traitements necessaires pour lister les contacts')
})

app.get('/contact/ajouter', (req,res)=>{
  //Les traitements necessaires pour ajouter un contact
  res.send('Les traitements necessaires pour ajouter un contact')
})

app.get('/contact/:id/modifier', (req,res)=>{
  //Les traitements necessaires pour modifier un contact
  res.send('Les traitements necessaires pour modifier un contact dont
1\'id='+req.params.id)
})

app.get('/contact/:id/supprimer', (req,res)=>{
  //Les traitements necessaires pour supprimer un contact
  res.send('Les traitements necessaires pour supprimer le contact id='+req.params.id)
})

```

21. Créer une base de données nommée : « merndb » et la collection « contacts »

MongoDB est une base de données NoSQL très populaire qui stocke des données sous forme de documents JSON (BSON en interne). MongoDB est souvent utilisé dans des applications MERN (MongoDB, Express.js, React, Node.js) en tant que base de données pour stocker et récupérer des données. Le paquet Mongoose est une bibliothèque Node.js qui facilite l'interaction avec MongoDB en fournissant une couche d'abstraction plus élevée, des fonctionnalités de validation des données et de modélisation des données.

Voici une introduction sur l'utilisation de MongoDB avec Mongoose dans une application MERN :

1. Installation de MongoDB :

Pour utiliser MongoDB dans une application MERN, vous devez d'abord installer MongoDB sur votre serveur ou utiliser un service d'hébergement cloud MongoDB comme MongoDB Atlas.

2. Installation de Mongoose : Utilisez npm (ou yarn) pour installer le paquet Mongoose dans votre application Node.js :

```
npm install mongoose
```

3. Configuration de la connexion à MongoDB : Dans votre application Node.js, configurez la connexion à la base de données MongoDB en utilisant Mongoose. Cela implique de fournir l'URL de connexion à MongoDB, par exemple, en utilisant l'URL de MongoDB Atlas ou une URL locale si MongoDB est installé localement.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/mydatabase', { useNewUrlParser: true,
useUnifiedTopology: true });
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'Erreur de connexion à MongoDB :'));
db.once('open', () => {
  console.log('Connecté à MongoDB');
});
```

4. Modélisation des données : Avec Mongoose, vous définissez des schémas et des modèles pour vos données. Les schémas décrivent la structure des données, tandis que les modèles correspondent à des collections dans la base de données.

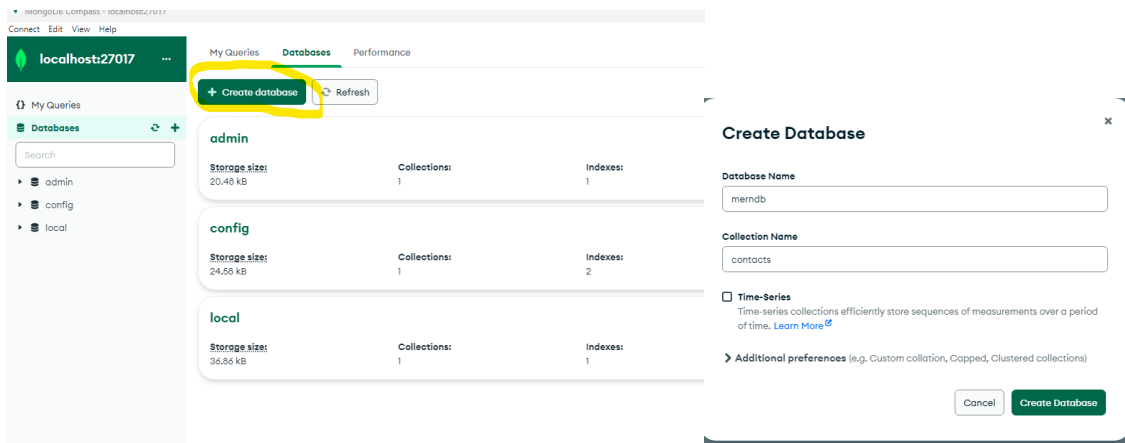
```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  username: String,
  email: String,
});
const User = mongoose.model('User', userSchema);
```

5. Utilisation des modèles dans l'application : Vous pouvez utiliser les modèles Mongoose pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur la base de données MongoDB. Par exemple, pour créer un nouvel utilisateur :

```
const newUser = new User({ username: 'john_doe', email: 'john@example.com' });
newUser.save((err, user) => {
  if (err) {
    console.error('Erreur lors de l\'enregistrement de l\'utilisateur :', err);
  } else {
    console.log('Utilisateur enregistré avec succès :', user);
  }
});
```

6. Intégration avec une application MERN : Vous pouvez utiliser les modèles Mongoose dans votre API Express.js pour gérer les données de votre application MERN. Dans votre application React, vous pouvez effectuer des requêtes HTTP vers votre API Express.js pour interagir avec la base de données MongoDB.

En résumé, MongoDB est une base de données NoSQL populaire utilisée dans les applications MERN, et Mongoose est une bibliothèque Node.js qui facilite la gestion des opérations de base de données MongoDB. Ils offrent une solution puissante et flexible pour stocker et gérer les données de votre application MERN.



22. Installer « mongoose »

```
npm i mongoose
```

23. Ajouter le code suivant

// Connexion au serveur db mongodb

```
const mongoose = require('mongoose')
mongoose.connect('mongodb://127.0.0.1:27017/merndb',{
  useNewUrlParser: true
})

const db = mongoose.connection;

db.on('error', console.error.bind(console, 'connexion error'))
db.once('open', function() {
  console.log('connexion avec succès ')
})
app.use(bodyParser.urlencoded({ extended: true }));
```

24. Model

Voici quelques exemples simples de schémas Mongoose pour vous montrer comment définir la structure de données dans MongoDB en utilisant Mongoose :

Exemple 1 : Schéma pour un modèle d'utilisateur

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  age: Number,
});
const User = mongoose.model('User', userSchema);
module.exports = User;
```

Dans cet exemple, nous définissons un schéma pour un modèle d'utilisateur avec trois champs : username, email, et age. Le champ username est de type String et requis. Le champ email est également de type String, requis, et doit être unique (pas de doublons autorisés). Le champ age est de type Number.

Exemple 2 : Schéma pour un modèle de produit

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

const productSchema = new mongoose.Schema({
  name: String,
  description: String,
  price: Number,
  category: {
    type: String,
    enum: ['Electronics', 'Clothing', 'Books', 'Home & Garden'],
  },
});

const Product = mongoose.model('Product', productSchema);

module.exports = Product;
```

Dans cet exemple, nous définissons un schéma pour un modèle de produit avec les champs name, description, price, et category. Le champ category est un champ de type String qui doit appartenir à l'une des valeurs spécifiées dans l'enum (énumération), ce qui signifie qu'il doit être l'un des types de catégories spécifiés.

Exemple 3 : Schéma pour un modèle de commentaire associé à un article

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

const commentSchema = new mongoose.Schema({
  text: String,
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User', // Référence au modèle User
  },
  article: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Article', // Référence au modèle Article
  },
});

const Comment = mongoose.model('Comment', commentSchema);

module.exports = Comment;
```

Dans cet exemple, nous définissons un schéma pour un modèle de commentaire avec les champs text, author, et article. Les champs author et article sont des références à d'autres modèles, "User" et "Article". Cela permet de créer des relations entre les commentaires, les utilisateurs et les articles.

Ces exemples simples de schémas Mongoose illustrent comment définir la structure de données pour différents modèles dans une application utilisant MongoDB et Mongoose. Vous pouvez personnaliser ces schémas en fonction des besoins spécifiques de votre application.

25. Créer un dossier Models sous la racine de l'application contenant le fichier contact.js

```
const mongoose = require('mongoose')
const ContactSchema = new mongoose.Schema({
  nom:{
    type:String,
    required:true,
  },
  tel:{
    type:String,
    required:true,
  }
}, { timestamps: true})

module.exports = mongoose.model('contacts', ContactSchema)
```

Plus des exemples :

<https://mongoosejs.com/docs/schematypes.html>

<https://mongoosejs.com/docs/guide.html#definition>

26. Ajouter ce code dans le fichier app.js (Point d'entrée : app ou bien index)

```
.....
const ContactModel = require('./Models/Contact')
.....
app.get('/contact/ajouter', (req, res) => {
  //Les traitements necessaires pour ajouter un contact
  const ContactObjet = {
    nom: "Kamel ABBASSI",
    tel: "98567890"
  }

  const contact = new ContactModel(ContactObjet)
  contact.save().then((contact) => {
    return res.status(200).json({message: 'Contact ajouté avec succès : '+contact})
  }).catch((err) => {
    res.status(400).json({message: 'Le Contact n\'est pas ajouté !'+err});
  });
})
```

27. Tester l'ajout simple et avec double

```
1 {
2   "nom": "kamel abbassi",
3   "tel": "8975451343",
4   "_id": "65492b12fe1c66174a6691c9",
5   "createdAt": "2023-11-06T18:06:10.676Z",
6   "updatedAt": "2023-11-06T18:06:10.676Z",
7   "__v": 0
8 }
```

```
1 {
2   "index": 0,
3   "code": 11000,
4   "keyPattern": {
5     "tel": 1
6   },
7   "keyValue": {
8     "tel": "89754512"
9   }
10 }
```

GET http://127.0.0.1:3000/contact/ajouter

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers (7) Test Results 200 OK 49 ms 471 B Save Response

Pretty Raw Preview Visualize JSON

```

1
2
3
{"message": "Contact ajout\u00e9 avec succ\u00e8s :{\n nom: 'ABBASSI Kamel',\n tel: '98567890',\n _id: new\n  ObjectId('\u0027651759be349f2f85d3e5672e\u0027'),\n createdAt: 2023-09-29T23:11:58.621Z,\n updatedAt:\n  2023-09-29T23:11:58.621Z,\n __v: 0\n}"}

```

```

_id: ObjectId('651751bbbe4b378ceeb6e3ef')
nom: "ABBASSI Kamel"
tel: "98567890"
createdAt: 2023-09-29T22:37:47.895+00:00
updatedAt: 2023-09-29T22:37:47.895+00:00
__v: 0

```

```

_id: ObjectId('651753f9be4b378ceeb6e3f1')
nom: "ABBASSI Kamel"
tel: "98567890"
createdAt: 2023-09-29T22:47:21.374+00:00
updatedAt: 2023-09-29T22:47:21.374+00:00
__v: 0

```

```

_id: ObjectId('651759737b02160431562f0a')
nom: "ABBASSI Kamel"
tel: "98567890"
createdAt: 2023-09-29T23:10:43.695+00:00
updatedAt: 2023-09-29T23:10:43.695+00:00
__v: 0

```

28. Passer des param\u00e8tres en URL

```

app.post('/contact/ajouter', (req, res) => {
  //Les traitements necessaires pour ajouter un contact

  const ContactObjet = {
    nom: req.body.nom,
    tel: req.body.tel
  }
  .....

```

POST http://127.0.0.1:3000/contact/ajouter

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

```

1
2
3
4
{"nom": "sami toumi",
 "tel": "6655665"}

```

```

_id: ObjectId('6517f63c7191cffd93245f97')
nom: "sami toumi"
tel: "6655665"
createdAt: 2023-09-30T10:19:40.601+00:00
updatedAt: 2023-09-30T10:19:40.601+00:00
__v: 0

```

```

_id: ObjectId('6517fb25b9547c2601afd02a')
nom: "sami toumi"
tel: "23456789"
createdAt: 2023-09-30T10:40:37.369+00:00
updatedAt: 2023-09-30T10:40:37.369+00:00
__v: 0

```

```

_id: ObjectId('6517fb56b9547c2601afd02c')
nom: "Walid siraji"
tel: "9854321"
createdAt: 2023-09-30T10:41:26.136+00:00
updatedAt: 2023-09-30T10:41:26.136+00:00
__v: 0

```

29. Afficher la liste des contacts

```
//Ajouter contact
```

```

app.get('/contact/lister', (req, res) => {
  //Les traitements necessaires pour lister les contacts
  ContactModel.find({}).exec().then((liste) => {
    return res.status(200).json({liste})
  }).catch((err) => {
    res.status(400).json({message: err});
  });
  //res.send('Les traitements necessaires pour lister les contacts')
})

```

```

"liste": [
  {
    "_id": "6517f63c7191cffd93245f97",
    "nom": "sami toumi",
    "tel": "6655665",
    "createdAt": "2023-09-30T10:19:40.601Z",
    "updatedAt": "2023-09-30T10:19:40.601Z",
    "__v": 0
  },
  {
    "_id": "6517fb25b9547c2601afd02a",
    "nom": "sami toumi",
    "tel": "23456789",
    "createdAt": "2023-09-30T10:40:37.369Z",
    "updatedAt": "2023-09-30T10:40:37.369Z",
    "__v": 0
  },
  {
    "_id": "6517fb56b9547c2601afd02c",
    "nom": "Walid siraji",
    "tel": "98654321",
    "createdAt": "2023-09-30T10:41:26.136Z",
    "updatedAt": "2023-09-30T10:41:26.136Z",
    "__v": 0
  }
]

```

30. Ajouter la méthode suivante pour supprimer un contact donné

```

app.get('/contact/:id/supprimer', (req, res) => {
  //Les traitements necessaires pour supprimer un contact

  ContactModel.findByIdAndDelete(req.params.id).exec().then((contactDeleted) => {
    return res.status(200).json({contactDeleted})
  }).catch((err) => {
    res.status(400).json({message: err});
  });

  //res.send('Les traitements necessaires pour supprimer le contact
  id='+req.params.id)
})

```



http://127.0.0.1:3000/contact/6517fb25b9547c2601afd02a/supprimer

GET



http://127.0.0.1:3000/contact/6517fb25b9547c2601afd02a/supprimer

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

Body Cookies Headers (7) Test Results

Pretty

Raw

Preview

Visualize

JSON



```

1  {
2    "contactDeleted": {
3      "_id": "6517fb25b9547c2601afd02a",
4      "nom": "sami toumi",
5      "tel": "23456789",
6      "createdAt": "2023-09-30T10:40:37.369Z",
7      "updatedAt": "2023-09-30T10:40:37.369Z",
8      "__v": 0
9    }
10 }

```

31. Ajouter la méthode suivante pour modifier un contact donné

```
app.get('/contact/:id/modifier', (req, res) => {
  //Les traitements nécessaires pour modifier un contact
  ContactModel.findByIdAndUpdate(req.params.id, {
    nom: 'Naceur Amairi',
    tel: '98456123'
  }).exec().then((contactUpdated) => {
    return res.status(200).json({contactUpdated})
  }).catch((err) => {
    res.status(400).json({message: err});
  });
  //res.send('Les traitements nécessaires pour modifier un contact dont
  1\ 'id='+req.params.id)
})
```

URL: http://127.0.0.1:3000/contact/6517fb56b9547c2601afd02c/modifier

Method: GET

Response (JSON):

```
{
  "contactUpdated": {
    "_id": "6517fb56b9547c2601afd02c",
    "nom": "Walid siraji",
    "tel": "98664321",
    "createdAt": "2023-09-30T18:41:26.136Z",
    "updatedAt": "2023-09-30T18:41:26.136Z",
    "__v": 0
  }
}
```

32. Paramétrer Postman pour envoyer les nouvelles données dans l'URL

URL: http://127.0.0.1:3000/contact/6517fb56b9547c2601afd02c/modifier?nom=walid fantazi&tel=3456789

Method: GET

Query Params:

Key	Value
<input checked="" type="checkbox"/> nom	walid fantazi
<input checked="" type="checkbox"/> tel	3456789

Response (JSON):

```
{
  "contactUpdated": {
    "_id": "6517fb56b9547c2601afd02c",
    "nom": "Walid siraji",
    "tel": "8768768",
    "createdAt": "2023-09-30T18:41:26.136Z",
    "updatedAt": "2023-10-01T17:06:12.699Z",
    "__v": 0
  }
}
```

33. Changer le code précédent pour récupérer les nouvelles données à partir de body avec la méthode « Post »

```
app.post('/contact/:id/modifier', (req, res) => {
  // Les traitements nécessaires pour modifier un contact
  ContactModel.findByIdAndUpdate(req.params.id, {
    nom: req.body.nom,
    tel: req.body.tel
  }).exec().then((contactUpdated) => {
    return res.status(200).json({contactUpdated});
  }).catch((err) => {
    res.status(400).json({message: err});
  });
  // res.send('Les traitements nécessaires pour modifier un contact dont
  id ' + req.params.id)
})
```

HTTP http://127.0.0.1:3000/contact/6517fb56b9547c2601afd02c/modifier

POST http://127.0.0.1:3000/contact/6517fb56b9547c2601afd02c/modifier

Params Authorization Headers (8) Body Pre-request Script Tests Settings

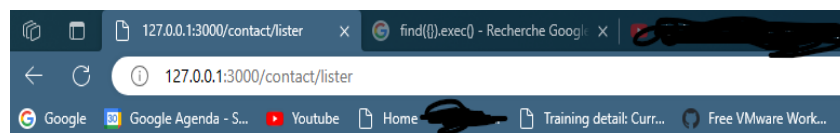
none form-data x-www-form-urlencoded raw binary JSON

```
1 {
2   "nom": "Walid siraji",
3   "tel": "8768768"
4 }
5
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "contactUpdated": {
3     "_id": "6517fb56b9547c2601afd02c",
4     "nom": "Naceur AmairiA",
5     "tel": "98456123",
6     "createdAt": "2023-09-30T10:41:26.136Z",
7     "updatedAt": "2023-10-01T16:49:08.344Z",
8     "__v": 0
9   }
10 }
```



```
1 {
2   "liste": [
3     {
4       "_id": "6517fb56b9547c2601afd02c",
5       "nom": "Walid siraji",
6       "tel": "8768768",
7       "createdAt": "2023-09-30T10:41:26.136Z",
8       "updatedAt": "2023-10-01T16:52:14.112Z",
9       "__v": 0
10    }
11  ]
12 }
```


34. Ajouter le code suivant pour valider les données envoyées

```
app.post('/contact/ajouter', (req, res) => {

  function isValidPhoneNumber(phoneNumber) {
    const phoneRegex = /^[2-9]\d{7}$/; // Format de numéro : 8 chiffres
    return phoneRegex.test(phoneNumber);
  }

  function isValidName(nom) {
    const nomRegex = /^[a-zA-Z\s]{2,20}$/;
    return nomRegex.test(nom);
  }

  // Vérifier si le nom est présent
  if (!isValidName(req.body.nom)) {
    return res.status(400).json({ message: 'Le nom est requis. min=2 et max=20' });
  }

  // Vérifier si le numéro de téléphone est présent et est un numéro valide
  if (!isValidPhoneNumber(req.body.tel)) {
    return res.status(400).json({ message: 'Le numéro de téléphone est invalide.'
  });
  }

  const object = {
    nom: req.body.nom,
    tel: req.body.tel,
  }
  const contact = new ContactModel(object);
  contact.save().then((contact) => {
    return res.status(200).send(contact);
  }).catch((error) => {
    //console.log(error);
    return res.status(400).send(error);
  });
});
```

35. Sécuriser l'API (Reporter à la fin de la formation)

L'authentification via une API en utilisant Node.js et Express.js peut être mise en œuvre de différentes manières. Une méthode courante consiste à utiliser JSON Web Tokens (JWT) pour gérer l'authentification. Voici un exemple simple de mise en place de l'authentification avec JWT dans une application Node.js avec Express.js :

3. Installez les dépendances nécessaires (Express, jsonwebtoken, body-parser) :

```
npm install jsonwebtoken

const express = require('express');
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;
```

// Clé secrète pour la création et la vérification des JWT

```
const secretKey = 'votreclésecrete';
```

// Middleware pour analyser le corps des requêtes au format JSON

```
app.use(bodyParser.json());
```

// Middleware pour gérer l'authentification

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Dans un véritable cas d'utilisation, vous vérifieriez les informations
  // d'authentification ici
  // Si l'authentification réussit, vous pouvez générer un JWT
  if (username === 'utilisateur' && password === 'motdepasse') {
    const token = jwt.sign({ username }, secretKey, { expiresIn: '1h' });

    res.json({ token });
  } else {
    res.status(401).json({ message: 'L\'authentification a échoué' });
  }
});
```

http://127.0.0.1:3000/contact/lister

GET http://127.0.0.1:3000/contact/lister

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

Key	Value
Key	Value

Body Cookies Headers (7) Test Results Status: 401 Unaut

Pretty Raw Preview Visualize JSON

```
1
2  "message": "Aucun token fourni"
3
```

// Middleware pour protéger les routes nécessitant une authentification

```
app.get('/contact/lister', (req, res) => {

  // Vérification du JWT dans l'en-tête Authorization
  const token = req.headers.authorization;

  if (!token) {
    res.status(401).json({ message: 'Aucun token fourni' });
    return;
  }

  jwt.verify(token, secretKey, (err, decoded) => {
    if (err) {
      res.status(401).json({ message: 'Token non valide' });
    } else {
      res.json({ message: 'Ressource sécurisée', user: decoded.username });
    }
  });

  ContactModel.find({}).exec().then((liste) => {

    console.log(liste);
    return res.status(200).json(liste);
  })

  .catch((error) => {
    return res.send(error);
  })

});

})
```

6. Vous pouvez utiliser un outil comme Postman ou cURL pour tester votre application. Voici comment vous pouvez vous authentifier et accéder à une ressource sécurisée :

a. Faites une requête POST à `http://localhost:3000/login` avec un corps JSON contenant le nom d'utilisateur et le mot de passe :

```
{
  "username": "utilisateur",
  "password": "motdepasse"
}
```

b. Vous recevrez un token JWT en réponse.

c. Copiez ce token et faites une requête GET à `http://localhost:3000/secure`, en ajoutant l'en-tête `Authorization` avec la valeur `Bearer [votre-token]`, où `[votre-token]` est le JWT que vous avez reçu.

Vous pouvez personnaliser davantage votre système d'authentification, notamment en stockant les informations d'utilisateur dans une base de données et en vérifiant les informations d'authentification dans votre middleware d'authentification. Ce qui précède est un exemple très simplifié pour illustrer le concept d'authentification avec Express.js et JWT.

Débogage

1. Erreur :

connexion error MongoServerError: connect ECONNREFUSED ::1:27017

Solution :

If you are using latest nodejs (v17.x) , then try updating mongodb url from localhost to 127.0.0.1

2. Erreur :

TypeError: Cannot read properties of undefined (reading 'nom')

Solution :

Ajouter le code en jaune

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

```
const app = express()
```

```
const port = 3000
```

```
app.use(express.json());
```

.....

Ou bien ajouter

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

Application serveur : Controller- Router

1. Organiser notre application backend

Dans une application MERN (MongoDB, Express, React, Node.js), les contrôleurs et les routes sont des composants essentiels pour organiser et gérer les fonctionnalités de l'application. Voici un aperçu de leurs rôles respectifs :

Contrôleurs (Controllers) :

Les contrôleurs sont des modules qui contiennent la logique métier de votre application. Ils sont responsables de la gestion des requêtes HTTP et de la manipulation des données en fonction de ces requêtes. Les rôles des contrôleurs dans une application MERN sont les suivants :

Recevoir les données d'une requête HTTP (paramètres de requête, corps de la requête, etc.).

Traiter les données reçues, telles que la validation, la transformation et la manipulation.

Effectuer des opérations sur la base de données, comme la création, la lecture, la mise à jour et la suppression (CRUD).

Appliquer la logique métier, telle que la gestion des règles de sécurité, la gestion des autorisations et la gestion des transactions.

Générer une réponse appropriée à renvoyer au client, généralement sous forme de données JSON.

Exemple de contrôleur dans une API Node.js :

```
Copy code
const User = require('../models/User');

const createUser = async (req, res) => {
  try {
    const newUser = new User(req.body);
    const savedUser = await newUser.save();
    res.status(201).json(savedUser);
  } catch (error) {
    res.status(400).json({ error: 'Échec de la création de l'utilisateur' });
  }
};

const getUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: 'Erreur lors de la récupération des utilisateurs' });
  }
};

module.exports = {
  createUser,
  getUsers,
};
```

Routes (Routes) : Les routes définissent les points de terminaison (endpoints) de votre API et spécifient comment les requêtes HTTP doivent être gérées. Elles associent les URL aux contrôleurs appropriés. Les rôles des routes dans une application MERN sont les suivants :

Définir les chemins d'accès URL auxquels les utilisateurs peuvent accéder pour interagir avec votre API.

Associer chaque chemin d'accès à un ou plusieurs contrôleurs pour traiter les requêtes.

Gérer les méthodes HTTP (GET, POST, PUT, DELETE, etc.) et les paramètres des requêtes, le cas échéant.

Faciliter la navigation et l'accès aux différentes fonctionnalités de l'application.

Exemple de routes avec Express.js :

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

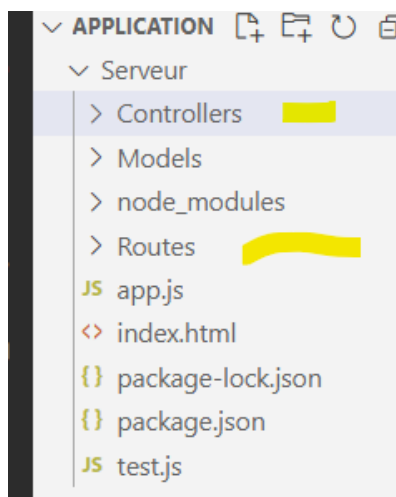
// Définition des routes pour la gestion des utilisateurs
router.post('/users', userController.createUser);
router.get('/users', userController.getUsers);

module.exports = router;
```

Dans cet exemple, les routes définissent deux points de terminaison : /users pour la création d'utilisateurs et la récupération d'utilisateurs. Ces routes sont associées aux contrôleurs correspondants.

En résumé, les contrôleurs gèrent la logique métier de votre application, tandis que les routes définissent les points d'accès pour interagir avec cette logique. En utilisant ces deux composants, vous pouvez organiser proprement votre application MERN et faciliter la gestion des requêtes HTTP et la gestion des données.

2. Créer les deux dossiers « Controllers » et « Routes » sous le dossier serveur



Le dossier **Controllers** contient les définitions de fonctions : ajouter, modifier, lister et supprimer

Le dossier **Routes** contient les chemins d'accès ou ressources (les routes = URL)

Remarques :

- Chaque ensemble des actions qui se tourne sur un seul objet sera dans un seul contrôleur.

Créer un nouveau fichier « **Controllers/ContactController.js** » avec le code suivant :

```
const ContactModel = require('../Models/contact');
//Ajouter contact
exports.ajouterContact= (req, res) => {

  function isValidPhoneNumber(phoneNumber) {
    const phoneRegex = /^[2-9]\d{7}$/; // Format de numéro : 8 chiffres
    return phoneRegex.test(phoneNumber);
  }

  function isValidName(nom) {
    const nomRegex = /^[a-zA-Z]{2,20}$/;
    return nomRegex.test(nom);
  }

  // Vérifier si le nom est présent
  if (!isValidName(req.body.nom)) {
    return res.status(400).json({ message: 'Le nom est requis. min=2 et max=20' });
  }

  // Vérifier si le numéro de téléphone est présent et est un numéro valide
  if (!isValidPhoneNumber(req.body.tel)) {
    return res.status(400).json({ message: 'Le numéro de téléphone est invalide.' });
  }

  const object = {
    nom: req.body.nom,
    tel: req.body.tel,
  }
  const contact = new ContactModel(object);
  contact.save().then((contact) => {
    return res.status(200).send(contact);
  }).catch((error) => {
    //console.log(error);
    return res.status(400).send(error);
  });
}
//Fin ajouter contact
```

3. Créer un nouveau fichier « Routes/Contact.js » avec le code suivant :

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

const controller = require('../Controllers/ContactController');
router.post('/contact/ajouter', controller.ajouterContact);

//Utilisation des routes à l'exterieur
module.exports= router;
```

4. Ajouter dans index.js le code suivant

```
const contactRouter= require('../Routers/Contact');
app.use('/', contactRouter);
```

5. Tester l'url « contact/ajouter »

HTTP <http://localhost:3000/contact/ajouter> Save

POST <http://localhost:3000/contact/ajouter> Send

Params Auth Headers (8) **Body** Pre-req. Tests Settings Cooki

x-www-form-urlencoded

	Key	Value	Bulk Ed
<input checked="" type="checkbox"/>	nom	walid ben salah	
<input checked="" type="checkbox"/>	tel	21626358	
	Key	Value	

Body 200 OK 14 ms 397 B Save Respons

Pretty Raw Preview Visualize JSON ≡

```
1 {
2   "nom": "walid ben salah",
3   "tel": "21626358",
4   "_id": "654cbafdf092b9d628dfc706",
5   "createdAt": "2023-11-09T10:57:01.444Z",
6   "updatedAt": "2023-11-09T10:57:01.444Z",
7   "__v": 0
}
```

6. Dans ContactController.js ajouter le code suivant :

```
exports.listerContact=(req,res)=>{
  //Les traitements necessaires pour lister les contacts
  ContactModel.find({}).exec().then((liste)=>{
    return res.status(200).json({liste})
  }).catch((err) => {
    res.status(400).json({message:err});
  });

  //res.send('Les traitements necessaires pour lister les contacts')
}
```

7. Ajouter le code suivant dans le fichier route.js

```
router.get('/contact/lister', controller.listerContact);
```


8. Tester la route « contact/lister »

🔗 <http://localhost:3000/contact/ajouter> 📄 Sav

GET ⌵ Send

Params Auth Headers (8) Body ● Pre-req. Tests Settings Cooki

Query Params

	Key	Value	Bulk Edit

Body ⌵ 🌐 200 OK 61 ms 711 B Save Response

Pretty Raw Preview Visualize JSON ⌵ 🔍

```

1  [
2    {
3      "_id": "654cb92516c0e8ce0f6323d2",
4      "nom": "walid",
5      "tel": "21626388",
6      "createdAt": "2023-11-09T10:49:09.323Z",
7      "updatedAt": "2023-11-09T10:49:09.323Z",
8      "__v": 0
9    },
10   {
11     "_id": "654cb9fdf092b9d628dfc704",
12     "nom": "walid ben mo",
13     "tel": "21626356",
14     "createdAt": "2023-11-09T10:52:45.319Z",

```

9. Ajouter ce code dans le controller

```

//Modifier contact
exports.modifierContact = (req, res) => {
  const newData = {
    nom: req.body.nom,
    tel: req.body.tel
  }
  ContactModel.findByIdAndUpdate(req.params.id,
newData).exec().then((contactUpdated) => {
    res.status(200).send(contactUpdated);
  }).catch((error) => {
    res.status(400).send(error);
  });
}
// fin modifier contact

```

10. Ajouter ce code dans Routers/contact.js

```

router.put('/contact/:id/modifier',controller.modifierContact);

```

11. Tester la route « contact/modifier »

HTTP client interface showing a PUT request to `http://127.0.0.1:3000/contact/65492b12fe1c66174a6691c9/modifier?nom=wali&tel=778687`.

The request body is set to `x-www-form-urlencoded` with the following parameters:

Key	Value	Bulk Edit
<input checked="" type="checkbox"/> nom	sami ousleti	
<input checked="" type="checkbox"/> tel	98123456	
Key	Value	

The response status is 200 OK, 16 ms, 387 B. The response body is:

```
{ "_id": "654cb92516c0e8ce0f6323d2", "nom": "iyuyi", "tel": "78787878", "createdAt": "2023-11-09T10:49:09.323Z", "updatedAt": "2023-11-09T11:39:58.665Z", "__v": 0 }
```

12. Ajouter le code suivant dans le controller

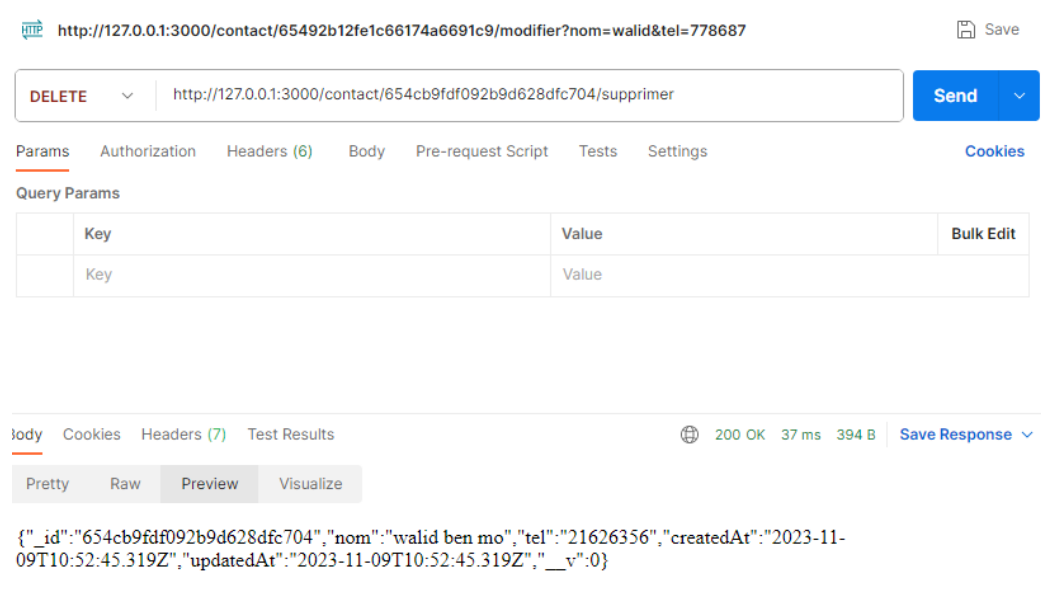
```
//supprimer contact
exports.supprimerContact = (req, res) => {
  //Les traitements necessaires pour supprimer un contact

  ContactModel.findByIdAndDelete(req.params.id).exec().then((contactDeleted) => {
    return res.status(200).send(contactDeleted);
  }).catch((error) => {
    return res.status(400).send(error);
  });
}
//fin supprimer contact
```

13. Ajouter le code suivant dans Routes/router.js

```
router.delete('/contact/:id/supprimer', controller.supprimerContact);
```

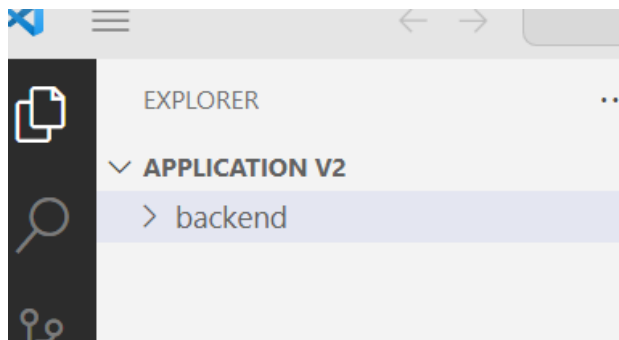
14. Tester l'url « contact/supprimer »



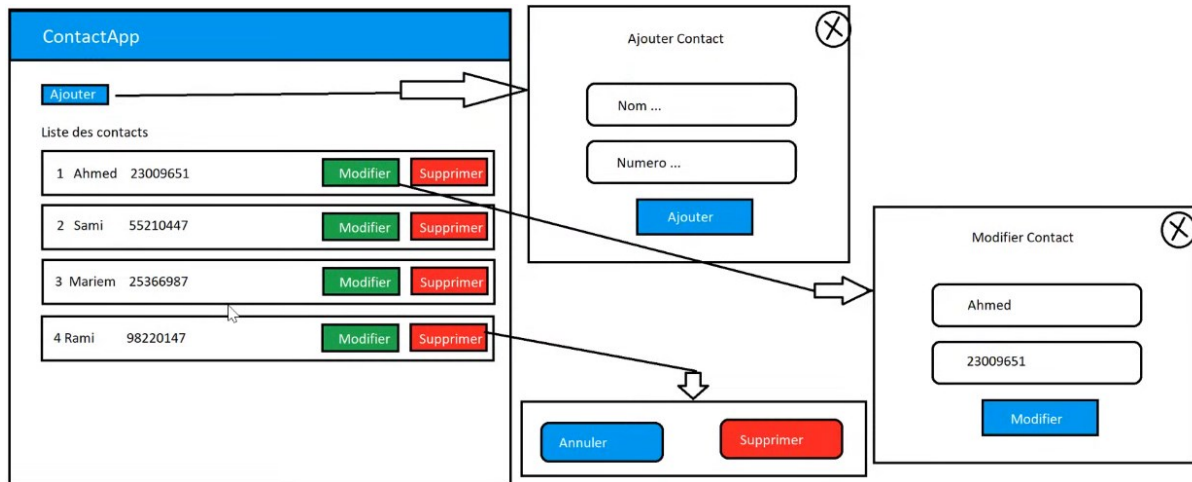
The screenshot shows a REST client interface with a DELETE request to `http://127.0.0.1:3000/contact/65492b12fe1c66174a6691c9/modifier?nom=walid&tel=778687`. The response is a 200 OK status with a response time of 37 ms and a body size of 394 B. The response body is a JSON object:

```
{ "_id": "654cb9fdf092b9d628dfc704", "nom": "walid ben mo", "tel": "21626356", "createdAt": "2023-11-09T10:52:45.319Z", "updatedAt": "2023-11-09T10:52:45.319Z", "__v": 0 }
```

15. Regrouper tous les fichiers de l'application dans un dossier nommé « backend »



Application Client : ReactJS



1. Créer un dossier « frontend » qui va recevoir l'application client qui va consomme les APIs développés coté serveur (Backend).

Lancer la commande suivante : ***npx create-react-app frontend***

```
C:\Users\kamel\Desktop\Application>npx create-react-app frontend
Creating a new React app in C:\Users\kamel\Desktop\Application\frontend.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

added 1458 packages in 2m
241 packages are looking for funding
```

2. Lancer les deux commandes suivantes :

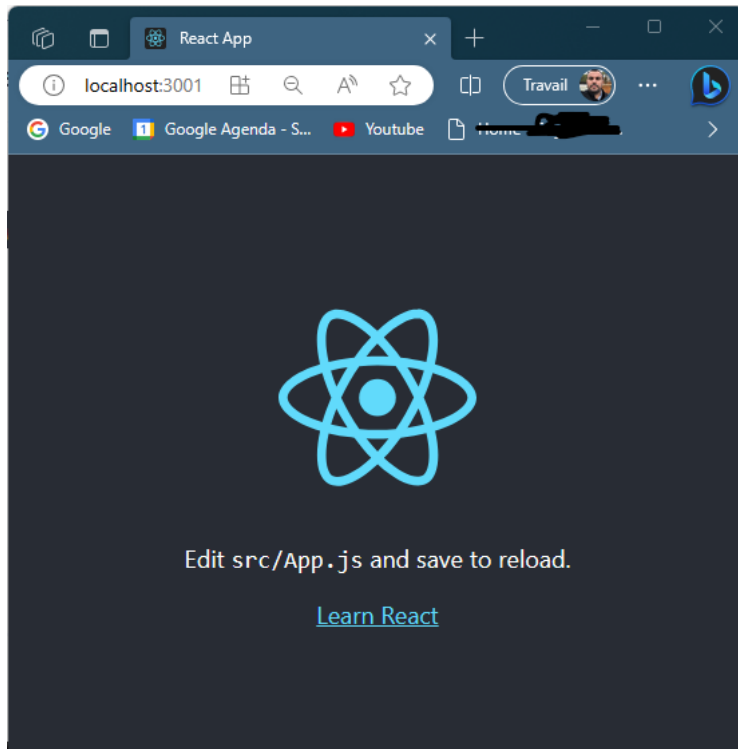
```
cd frontend
npm start
```

```
C:\Users\kamel\Desktop\Application\frontend>npm start

> frontend@0.1.0 start
> react-scripts start
? Something is already running on port 3000.

Would you like to run the app on another port instead? » (Y/n)
```

3. Valider avec la touche entre



4. Ajouter le Framework CSS Bootstrap

<https://react-bootstrap.github.io/docs/getting-started/introduction>

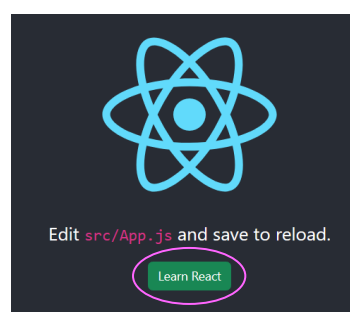
```
npm install react-bootstrap bootstrap
```

5. Dans le fichier app.js ajouter le code suivant :

```
import 'bootstrap/dist/css/bootstrap.min.css'  
import { Button } from 'react-bootstrap';
```

6. Entourer learn react par le code suivant :

```
<Button as="a" variant="success">  
  Learn React  
</Button>
```



7. Exemples avec reactJS

Un projet React et ajouter trois composants avec des tableaux HTML et des images. Voici les étapes :

Étape 1 : Création d'un projet React

Ouvrez votre terminal.

Exécutez la commande suivante pour créer un nouveau projet React avec Create React App (CRA). Remplacez nom-du-projet par le nom que vous souhaitez donner à votre projet :

```
npx create-react-app nom-du-projet
```

Une fois la création du projet terminée, accédez au répertoire de votre projet en utilisant la commande `cd nom-du-projet`.

Étape 2 : Ajout de composants avec tableaux HTML et images

Dans le répertoire `src`, créez un nouveau fichier nommé `Header.js` pour le composant Header :

Header.js :

```
import React from 'react';

function Header() {
  return (
    <header>
      <h1>Mon Application React</h1>
      <table>
        <thead>
          <tr>
            <th>Nom</th>
            <th>Âge</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>John Doe</td>
            <td>30</td>
          </tr>
          <tr>
            <td>Jane Smith</td>
            <td>25</td>
          </tr>
        </tbody>
      </table>
    </header>
  );
}

export default Header;
```

Créez un autre fichier nommé `Content.js` pour le composant Content :

Content.js

```
import React from 'react';

function Content() {
  return (
    <div>
      <h2>Contenu de l'application</h2>
      <p>Bienvenue dans la section de contenu !</p>
      
    </div>
  );
}

export default Content;
```

Modifiez le fichier src/App.js pour inclure les composants Header et Content :

App.js :

```
import React from 'react';
import Header from './Header'; // Import du composant Header
import Content from './Content'; // Import du composant Content

function App() {
  return (
    <div>
      <Header /> {/* Inclusion du composant Header */}
      <Content /> {/* Inclusion du composant Content */}
    </div>
  );
}

export default App;
```

Étape 3 : Exécution de l'application React

```
npm start
```

Ouvrez votre navigateur et accédez à <http://localhost:3000>.

Vous verrez votre composant App affichant le contenu des composants Header et Content. Le composant Header affiche un tableau HTML avec des noms et des âges, tandis que le composant Content affiche une image de démonstration.

Ce projet React vous permet d'ajouter des éléments visuels tels que des tableaux HTML et des images à vos composants pour rendre votre application plus riche en contenu. Vous pouvez personnaliser davantage ces éléments en fonction des besoins de votre application.

8. Essayer d'appliquer ce thème dans l'application client (Frontend)

Contacts Manager

V1.0

La liste des contacts

#	Nom	Tél	Actions
1	Mark	66585587	<div>Modifier</div> <div>supprimer</div>
2	Jacob	867857456	<div>Modifier</div> <div>supprimer</div>

Ajouter un contact

Nom:

Tél

Ajouter

Voici un exemple simple pour comprendre l'utilisation du hook `useState` en React. Dans cet exemple, nous allons créer un composant qui permet à l'utilisateur d'augmenter un compteur en appuyant sur un bouton.

Counter.js

```
import React, { useState } from 'react';

function Counter() {
  // Déclarez une variable d'état appelée "count" avec une valeur initiale de 0
  const [count, setCount] = useState(0);

  // Fonction pour augmenter le compteur
  const increment = () => {
    setCount(count + 1); // Met à jour la valeur de "count"
  };

  return (
    <div>
      <h1>Compteur</h1>
      <p>Valeur du compteur : {count}</p>
      <button onClick={increment}>Incrémenter</button>
    </div>
  );
}

export default Counter;
```

9. Appel de ce composant dans le fichier app.js

```
import Counter from './counter';
.....
<div className="App">
  <Counter />
.....
```

Dans cet exemple :

Nous utilisons le hook **useState** en important **useState** depuis React.

Nous déclarons une variable d'état appelée `count` avec une valeur initiale de 0 en utilisant `useState(0)`.

Nous définissons une fonction `increment` qui est appelée lorsque l'utilisateur clique sur le bouton. Cette fonction utilise `setCount` pour mettre à jour la valeur de `count` en ajoutant 1 à sa valeur actuelle.

Dans la méthode **`render`**, nous affichons la valeur actuelle de `count` à l'écran. Chaque fois que l'utilisateur clique sur le bouton "Incrémenter", la valeur de `count` est mise à jour et le composant est automatiquement réexécuté pour refléter cette modification.

En utilisant le hook `useState`, vous pouvez gérer l'état dans vos composants React de manière simple et efficace. Cela vous permet de rendre vos composants réactifs en fonction des données et des interactions de l'utilisateur. Dans cet exemple, nous avons créé un simple compteur, mais vous pouvez l'appliquer à des scénarios plus complexes dans vos applications React.

10. Voilà le code source de fichier app.js

```
import './App.css';
import 'bootstrap/dist/css/bootstrap.min.css';
import { useState } from 'react';

import {Modal, Navbar,Nav, Container, Row,Col, Table, Form, Button} from 'react-bootstrap';

function App() {

  const [edit, showEdit] = useState(false);
  const handleCloseEdit = () => showEdit(false);
  const handleShowEdit = () => showEdit(true);

  const [destroy, showDestroy] = useState(false);
  const handleCloseDestroy = () => showDestroy(false);
  const handleShowDestroy = () => showDestroy(true);

  return (
    <div className="App">
      <Navbar bg="primary" data-bs-theme="dark">
        <Container>
          <Navbar.Brand href="#home">Contacts Manager</Navbar.Brand>
          <Nav className="me-auto">
            <Nav.Link href="#home">V1.0</Nav.Link>
          </Nav>
        </Container>
      </Navbar>

      <div className='container'>
        <div className='p-4'></div>

        <Row>
          <Col xs lg="8">
            <h2>La liste des contacts</h2>
            <Table striped bordered hover size="lg">
              <thead>
                <tr>
                  <th>#</th>
                  <th>Nom</th>
                  <th>Tél</th>
                  <th>Actions</th>
                </tr>
              </thead>
              <tbody>
                <tr>
                  <td>1</td>
                  <td>Mark</td>
                  <td>66585587</td>
                  <td>
                    <a href="/#" className='btn btn-success m-1' onClick={handleShowEdit}>
Modifier </a>
                    <a href="/#" className='btn btn-danger'
onClick={handleShowDestroy} >supprimer</a>
                  </td>
                </tr>
                <tr>
                  <td>2</td>
                  <td>Jacob</td>
                  <td>867857456</td>
                  <td><a href="/#" onClick={handleShowEdit} className='btn btn-success m-1'>Modifier</a>
                    <a href="/#" onClick={handleShowDestroy} className='btn btn-danger'>supprimer</a></td>
                </tr>
              </tbody>
            </Table>
          </Col>
        </Row>
      </div>
    </div>
  )
}
```

```

</Table>
</Col>
<Col xs lg="4">
  <h2>Ajouter un contact</h2>

  <Form>
    <Form.Group as={Row} className="mb-3" controlId="nom">
      <Form.Label column sm={2}>
        Nom:
      </Form.Label>
      <Col sm={10}>
        <Form.Control type="text" />
      </Col>
    </Form.Group>

    <Form.Group as={Row} className="mb-3" controlId="tel">
      <Form.Label column sm={2}>
        Tél
      </Form.Label>
      <Col sm={10}>
        <Form.Control type="tel" />
      </Col>
    </Form.Group>

    <Button variant="primary" type="submit">
      Ajouter
    </Button>
  </Form>

</Col>
</Row>
</div>

{/** Model edit */}

<Modal show={edit} onHide={handleCloseEdit}>
  <Modal.Header closeButton>
    <Modal.Title>Modifier contact</Modal.Title>
  </Modal.Header>
  <Modal.Body>

    <Form>
      <Form.Group as={Row} className="mb-3" controlId="nom">
        <Form.Label column sm={2}>
          Nom:
        </Form.Label>
        <Col sm={10}>
          <Form.Control type="text" />
        </Col>
      </Form.Group>

      <Form.Group as={Row} className="mb-3" controlId="tel">
        <Form.Label column sm={2}>
          Tél
        </Form.Label>
        <Col sm={10}>
          <Form.Control type="tel" />
        </Col>
      </Form.Group>

    </Form>

    </Modal.Body>
    <Modal.Footer>
      <Button variant="secondary" onClick={handleCloseEdit}>
        Close
      </Button>
      <Button variant="primary" onClick={handleCloseEdit}>
        Save Changes
      </Button>
    </Modal.Footer>
  </Modal>

```

```

    </Modal>

    /** Model supprimer */
    <Modal show={destroy} onHide={handleCloseDestroy}>
      <Modal.Header closeButton>
        <Modal.Title>Supprimer contact</Modal.Title>
      </Modal.Header>
      <Modal.Body>
        <p>Voulez vraiment supprimer ce contact ?</p>
      </Modal.Body>
      <Modal.Footer>
        <Button variant="secondary" onClick={handleCloseDestroy}>
          Close
        </Button>
        <Button variant="primary" onClick={handleCloseDestroy}>
          Save Changes
        </Button>
      </Modal.Footer>
    </Modal>

  </div>
);
}

export default App;

```

11. Ajouter les fenetre pop-up (Modal bootstrap) pour les deux boutons modifier et supprimer

Source : <https://react-bootstrap.netlify.app/docs/components/modal/>

```

.....
<a href='#' className='btn btn-success m-1' onClick={handleShowEdit}> Modifier
</a>
<a href='#' className='btn btn-danger' onClick={handleShowSupprimer}>
  >supprimer</a>
.....

```

```

    /** Model edit */
<Modal show={edit} onHide={handleCloseEdit}>
  <Modal.Header closeButton>
    <Modal.Title>Modifier contact</Modal.Title>
  </Modal.Header>
  <Modal.Body>

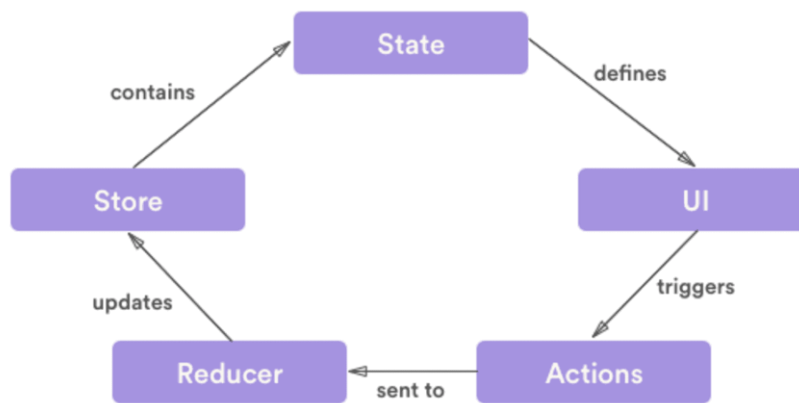
    <Form>
      <Form.Group as={Row} className="mb-3" controlId="nom">
        <Form.Label column sm={2}>
          Nom:
        </Form.Label>
        <Col sm={10}>
          <Form.Control type="text" />
        </Col>
      </Form.Group>

      <Form.Group as={Row} className="mb-3" controlId="tel">
        <Form.Label column sm={2}>
          Tél
        </Form.Label>
        <Col sm={10}>
          <Form.Control type="tel" />
        </Col>
      </Form.Group>
    </Form>

    </Modal.Body>
    <Modal.Footer>
      <Button variant="secondary" onClick={handleCloseEdit}>
        Close
      </Button>
      <Button variant="primary" onClick={handleCloseEdit}>
        Save Changes
      </Button>
    </Modal.Footer>
  </Modal>

  /** Model supprimer */
<Modal show={destroy} onHide={handleCloseDestroy}>
  <Modal.Header closeButton>
    <Modal.Title>Supprimer contact</Modal.Title>
  </Modal.Header>
  <Modal.Body>
    <p>Voulez vraiment supprimer ce contact ?</p>
  </Modal.Body>
  <Modal.Footer>
    <Button variant="secondary" onClick={handleCloseDestroy}>
      Close
    </Button>
    <Button variant="primary" onClick={handleCloseDestroy}>
      Save Changes
    </Button>
  </Modal.Footer>
</Modal>

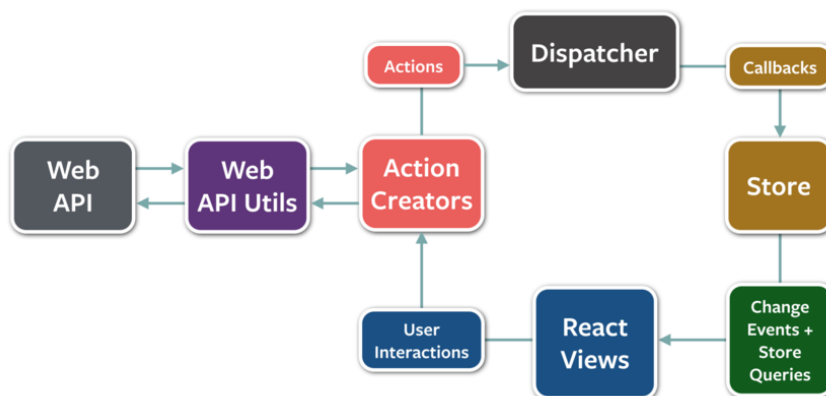
```



Redux

Le domaine du développement JavaScript est extrêmement vaste, offrant une pléthore de développeurs ainsi que de nombreux frameworks et outils. Lorsque vous entreprenez le développement d'une application, quel que soit le framework de rendu que vous choisissiez, la nécessité d'architecturer votre projet se fait rapidement sentir. Cela devient particulièrement évident lors de l'utilisation de frameworks de rendu de composants tels que React ou VueJS. Historiquement, ce besoin s'est manifesté de manière significative dans le contexte de React, incitant Facebook à ouvrir les sources de son outil Flux.

Le principe est le suivant :



Votre application spécifie les actions associées à chaque composant, lesquelles définissent l'état du composant stocké dans un magasin (store) pour maintenir la vue à jour. Cependant, l'inconvénient réside dans le fait qu'un magasin est attribué à chaque composant, ce qui peut être limitant dans certaines applications, même si cela fonctionne bien pour React. Pour remédier à cela, Dan Abramov a introduit Redux en juin 2015, simplifiant la gestion du magasin en consolidant tous les états dans un seul magasin pour l'ensemble de l'application. Ainsi, tous les composants peuvent accéder aux données de manière plus globale.

Dans une application React utilisant Redux, les dossiers `reducers`, `actions`, et la configuration du `store` jouent des rôles spécifiques dans l'organisation de votre code.

1. ****Dossier `reducers`**:**

Le dossier `reducers` contient les fonctions reducers. Un reducer est une fonction qui spécifie comment l'état de l'application change en réponse à une action. Il prend en entrée l'état actuel de l'application et une action, puis renvoie le nouvel état.

- ****Rôle****:

- Définir comment les données dans le store vont être mises à jour en réponse aux actions.
- Organiser la logique métier de votre application en gérant les différentes actions.

- ****Exemple**** (dans `reducers.js`):

```
const initialState = {
  count: 0,
};
const rootReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return {
        ...state,
        count: state.count + 1,
      };
    case 'DECREMENT':
      return {
        ...state,
        count: state.count - 1,
      };
    default:
      return state;
  }
};

export default rootReducer;
```

2. ****Dossier `actions`**:**

Le dossier `actions` contient les créateurs d'actions. Une action est un objet JavaScript qui décrit le type de changement que vous souhaitez effectuer dans votre application.

- ****Rôle****:

- Définir les actions possibles que votre application peut effectuer.
- Encapsuler la logique de création d'actions pour rendre le code plus lisible et réutilisable.

- ****Exemple**** (dans `actions.js`):

```
export const increment = () => ({
  type: 'INCREMENT',
});

export const decrement = () => ({
  type: 'DECREMENT',
});
```

3. ****Store****:

Le `store` est un objet central dans Redux qui détient l'état de votre application. Il est créé à partir de la fonction `createStore` de Redux et prend en compte le reducer principal de votre application.

- **Rôle**:

- Centraliser l'état de l'application.

- Fournir des méthodes pour accéder à l'état actuel (`getState`), pour le mettre à jour (`dispatch`), et pour écouter les changements (`subscribe`).

- **Exemple** (dans `index.js`):

```
import { createStore } from 'redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);
```

En résumé, ces dossiers sont organisés de manière à séparer les responsabilités dans votre application React Redux. Les reducers gèrent la logique métier, les actions décrivent les changements, et le store centralise l'état de l'application. Cette séparation facilite la maintenance, l'extension et la compréhension du code, surtout lorsque l'application devient plus complexe.

12. Installer Redux avec la commande suivante :

```
npm i redux
```

13. installer react-redux

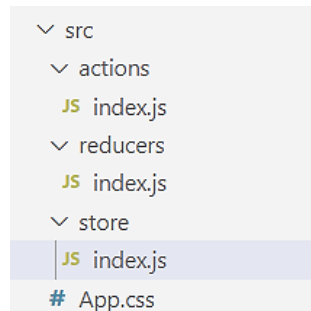
```
npm i react-redux
```

14. installer redux-thunk

```
npm i redux-thunk
```

15. Ajouter les dossiers suivants dans le dossier « src »

- actions
- reducers
- store



16. Dans chaque ces trois dossiers, créer un fichier « index.js »

17. Installer axios (package pour consommer les APIs)

```
npm i axios
```


18. Ajouter les fichiers suivants :

src/actions/constantes.js (Dans ce fichier enregistre les actions possibles que sera utilisées par les autres fichiers)

```
export const contactesConstants = {  
  GET_ALL_CONTACTS_REQUEST: 'GET_ALL_CONTACTS_REQUEST',  
  GET_ALL_CONTACTS_SUCCESS: 'GET_ALL_CONTACTS_SUCCESS',  
  GET_ALL_CONTACTS_FAILURE: 'GET_ALL_CONTACTS_FAILURE'  
}
```

src/actions/contact.actions.js

```
import axios from "axios";  
import { contactConstants } from "../constantes";  
  
export const listerContacts = () => {  
  return async dispatch => {  
    dispatch({ type: contactConstants.GET_ALL_CONTACTS_REQUEST })  
  
    try {  
      const res = await axios.get('http://127.0.0.1:3000/contact/lister')  
      if (res.status === 200) {  
        dispatch({  
          type: contactConstants.GET_ALL_CONTACTS_SUCCESS,  
          payload: { contacts: res.data } })  
        }  
      } catch (error) {  
        dispatch({  
          type: contactConstants.GET_ALL_CONTACTS_FAILURE,  
          payload: { error: error.res } })  
        }  
      }  
    }  
  }  
}
```

src/actions/index.js

```
export * from './contact.actions';
```

src/reducers/contact.reducer.js

```
import { contactConstants } from "../actions/constantes";

const initialState = {
  contacts : [],
  error: null
}

export default (state = initialState, action) => {
  switch (action.type) {
    case contactConstants.GET_ALL_CONTACTS_REQUEST:
      state = {
        ...state
      }
      break;
    case contactConstants.GET_ALL_CONTACTS_SUCCESS:
      state = {
        ...state,
        contacts: action.payload.contacts
      }
      break;

    case contactConstants.GET_ALL_CONTACTS_FAILURE:
      state = {
        ...state,
        error: action.payload.error
      }
      break;
  }
  return state;
}
```

19. Voici le code reduces/ index.js

```
import contactReducer from './contact.reducer';
import { combineReducers } from 'redux';

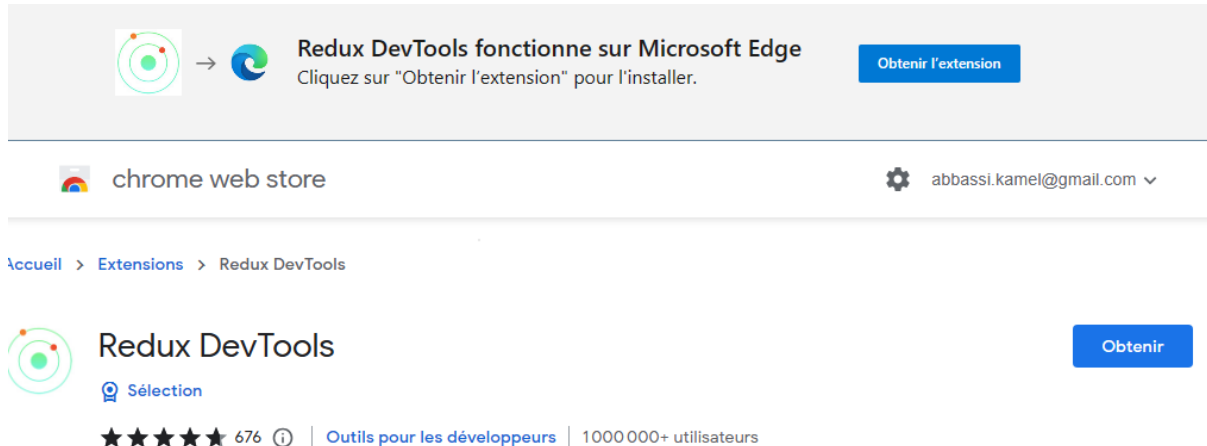
const rootReducer = combineReducers ({
  contact : contactReducer
})

export default rootReducer;
```

20. Voici le code store/index.js

```
import { applyMiddleware, legacy_createStore as createStore, legacy_createStore }
from "redux"
import rootReducer from "../reducers"
import thunk from "redux-thunk"
const store= legacy_createStore(
  rootReducer,
  applyMiddleware(thunk)
)
export default legacy_createStore;
```

21. Ajouter l'extension suivante :



chrome web store

accueil > Extensions > Redux DevTools

Redux DevTools

Sélection

★★★★★ 676 ⓘ | Outils pour les développeurs | 1000 000+ utilisateurs

Obtenir

22. Changer le code du fichier store/index.js comme suit :

```
import { createStore, applyMiddleware, compose } from 'redux'; // Import
createStore from 'redux'
import rootReducer from '../reducers';
import thunk from 'redux-thunk';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(
  rootReducer,
  composeEnhancers(applyMiddleware(thunk))
);

export default store;
```

Source : <https://github.com/zalmoxisus/redux-devtools-extension>

23. Installer cors dans le backend (serveur) pour que les autres applications peuvent consomment les APIs

```
npm i cors
```

24. Modifier le code du fichier Serveur/app.js comme suit

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

const app = express()
const port = 3000
app.use(express.json());

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

.....
```

25. Ajouter le code suivant dans le fichier Application\frontend\src\App.js

```
import './App.css';
import 'bootstrap/dist/css/bootstrap.min.css'
import { useEffect } from 'react';
import { useDispatch } from 'react-redux';
import { listerContacts } from './actions/contact.actions';

.....

function App() {

  .....

  const dispatch = useDispatch();

  useEffect(()=>{

    dispatch(ListerContacts());
    console.log('Test');
  });

  return (
    <div className="App ">
      <Navbar bg="primary" data-bs-theme="dark">
        .....
      </Navbar>
    </div>
  );
}

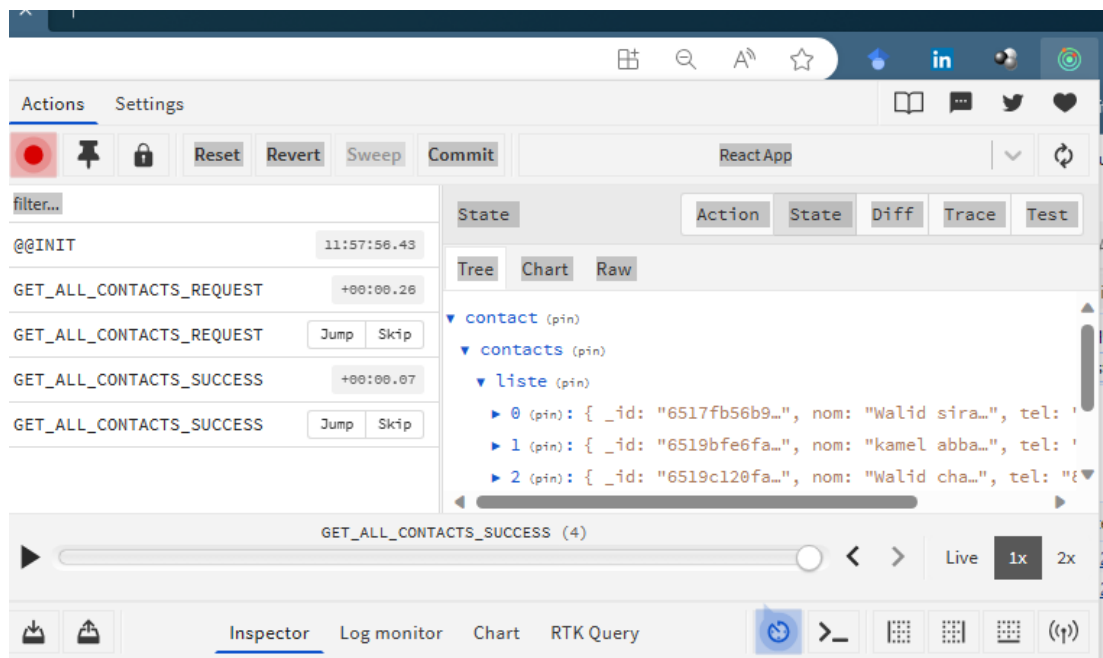
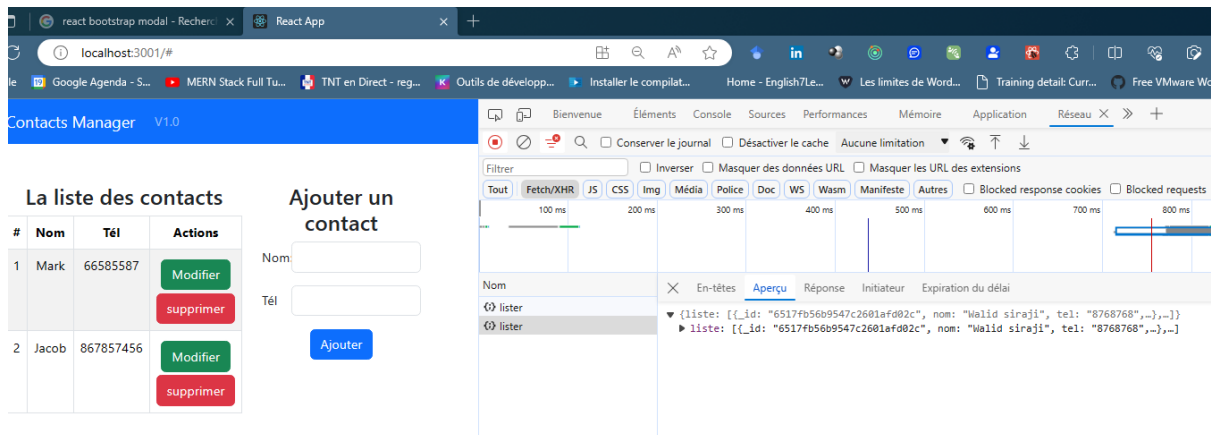
export default App;
```

26. Modifier le fichier « index.js » de l'application frontend

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { Provider } from 'react-redux';
import store from './store';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <React.StrictMode>
      <App />
    </React.StrictMode>
  </Provider>
);
reportWebVitals();
```

27. Tester le code



28. Changer le code du fichier Application\frontend\src\App.js

```

import './App.css';
import 'bootstrap/dist/css/bootstrap.min.css'
import { useEffect } from 'react';

import {Navbar,Nav, Container, Row,Col, Table, Form, Button, Modal} from 'react-
bootstrap';

import { ListerContacts } from './actions/contact.actions';
import { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux';

function App() {

  const dispatch = useDispatch();
  const contacts = useSelector(state => state.contact.contacts);

  useEffect(()=>{

    dispatch(ListerContacts());
    console.log('Test');
  });

  const [edit, setEdit] = useState(false);
  const handleCloseEdit = () => setEdit(false);
  const handleShowEdit = () => setEdit(true);

  const [supprimer, setSupprimer] = useState(false);
  const handleCloseSupprimer = () => setSupprimer(false);
  const handleShowSupprimer = () => setSupprimer(true);

  .....

  <Row>
    <Col xs lg="8">
      <h2>La liste des contacts</h2>
      { contacts ?
        <Table striped bordered hover size="lg">
          <thead>
            <tr>
              <th>#</th>
              <th>Nom</th>
              <th>Tél</th>
              <th>Actions</th>
            </tr>
          </thead>
          <tbody>

            { contacts.map( (contact, index)=>

              <tr key={index}>
                <td>{index}</td>
                <td>{contact.nom}</td>
                <td>{contact.tel}</td>
                .....
              </tr>

            )}

          </tbody>
        </Table>

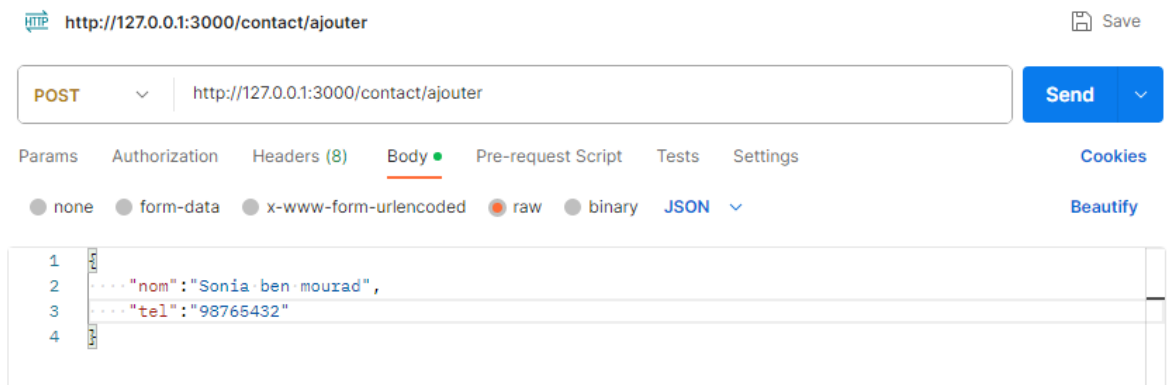
        : 'Aucun contact trouvé'
      }

    </Col>
    <Col xs lg="4">

```

```
export default App;
```

29. Tester l'ajout avec Postman



La liste des contacts

#	Nom	Tél	Actions
0	Walid siraji	66585587	<button>Modifier</button> <button>supprimer</button>
1	kamel abbassi	66585587	<button>Modifier</button> <button>supprimer</button>
2	Walid chaima	66585587	<button>Modifier</button> <button>supprimer</button>
3	Ayari chaima	66585587	<button>Modifier</button> <button>supprimer</button>
4	Ayari chaima	66585587	<button>Modifier</button> <button>supprimer</button>
5	Sonia ben mourad	66585587	<button>Modifier</button> <button>supprimer</button>

30. Mise en place du formulaire d'ajout

Ajouter un contact

Nom:

Tél

Ajouter

31. Définir ces trois variables id, nom et tel. Chaque variable est associée à une méthode de modification.

```
const [id, setId]= useState(''); //Utiliser ultérieurement
const [nom, setNom]= useState('');
const [tel, setTel]= useState('');
```

Le nom de la méthode commence par « set », ensuite le premier caractère de la variable est majuscule.

Les deux champs input (nom et tel) seront liés aux deux variables précédemment créées

```
<Form.Control type="text" value={nom} />
<Form.Control type="tel" value={tel} />
```

32. A chaque changement des valeurs des inputs, on va changer les valeurs des variables nom et tel

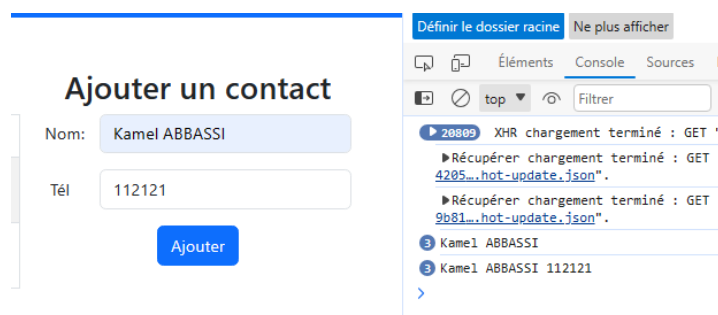
```
<Form.Control type="text" value={nom} onChange={ (e) => setNom(e.target.value)} />
<Form.Control type="text" value={tel} onChange={ (e) => setTel(e.target.value)} />
```

33. Le clic sur le bouton « ajouter » lance une fonction permet de prendre les valeurs des inputs et les mettent dans les statues

```
<Button variant="primary" type="submit" onClick={addContactEvent}>
  Ajouter
</Button>
```

34. Maintenant ajouter cette méthode en haut dans App.js

```
const addContactEvent= ()=>{
  console.log(nom,tel);
}
```



35. Ajouter les 3 actions suivantes pour gérer l'ajout d'un nouveau contact (Dans le fichier Application\frontend\src\actions\constantes.js)

```
ADD_CONTACT_REQUEST: 'ADD_CONTACT_REQUEST',
ADD_CONTACT_SUCCESS: 'ADD_CONTACT_SUCCESS',
ADD_CONTACT_FAILURE: 'ADD_CONTACT_FAILURE'
```


36. Ajouter la méthode addContactAction dans le fichier Application\frontend\src\actions\contact.actions.js

```
export const addContactAction=(data)=>{

  return async dispatch =>{
    console.log("dans action:"+data.tel);
    dispatch({type:constantesactions.ADD_CONTACT_REQUEST});

    var params = new URLSearchParams();
    params.append('nom', data.nom);
    params.append('tel', data.tel);
    try {
      const res = await
    axios.post('http://127.0.0.1:3000/contact/ajouter',params)

      if(res.status === 200){
        dispatch({
          type:constantesactions.ADD_CONTACT_SUCCESS,
          payload: {contact: res.data}})
      }

    } catch (error) {
      //console.log(error.response.data);
      alert(error.response.data.message);

      dispatch({
        type:constantesactions.ADD_CONTACT_FAILURE,
        payload:{ error:error.res}})
    }
  }
}
```

37. Ajouter le code suivant dans le fichier « Application\frontend\src\reducers\contact.reducer.js »

```

.....
const initialState = {
  contacts : [],
  error: null,
  contact: {}
}

.....
case contactConstants.ADD_CONTACT_REQUEST:
  state = {
    ...state,

  }
  break;

case contactConstants.ADD_CONTACT_SUCCESS:
  state = {
    ...state,
    contact: action.payload.contact
  }
  break;

case contactConstants.ADD_CONTACT_FAILURE:
  state = {
    ...state,
    error: action.payload.error
  }
  break;
.....

```

38. Modifier le fichier Application\frontend\src\App.js

```

import { ListerContacts, addContactAction } from '../actions/contact.actions';
.....
const addContactEvent= async ()=>{
  console.log(nom,tel);
  const data = {
    nom,
    tel
  }
  await dispatch(addContactAction(data));
  await dispatch(ListerContacts())
}

```

39. Ajouter un Bouton pour supprimer un contact donné.

Supprimer contact

×

Voulez vraiment supprimer ce contact ?

Close

Save Changes

40. Ajouter les constantes

```
DELETE_CONTACT_REQUEST: 'DELETE_CONTACT_REQUEST',
DELETE_CONTACT_SUCCESS: 'DELETE_CONTACT_SUCCESS',
DELETE_CONTACT_FAILURE: 'DELETE_CONTACT_FAILURE'
```

41. Ajouter l'action deleteContactAction ()

```
export const deleteContactAction=(id)=>{
  return async dispatch =>{
    dispatch({type: constantesactions.DELETE_CONTACT_REQUEST})
    try {
      console.log(id);
      const res = await
      axios.get('http://127.0.0.1:3000/contact/${id}/supprimer')
      console.log('res'+res);
      if(res.status === 200){
        console.log('data='+id);
        dispatch({
          type: constantesactions.DELETE_CONTACT_SUCCESS,
          payload: {contactCreated: res.id}
        })
      }
    } catch (error) {
      console.log('erreur est : '+ error);
      console.log(error.response.id);
      dispatch({
        type: constantesactions.DELETE_CONTACT_FAILURE,
        payload:{ error:error.response}
      })
    }
  }
}
```

42. Ajouter le code suivant dans le fichier contact.reducer.js

```
//Supprimer contact
case constantesactions.DELETE_CONTACT_REQUEST:
  state ={
    ...state,
  }
  break;

case constantesactions.DELETE_CONTACT_SUCCESS:
  state ={
    ...state,
    createdC: action.payload.message
  }
  break;

case constantesactions.DELETE_CONTACT_FAILURE:
  state ={
    ...state,
    error: action.payload.error
  }
  break;
// fin supprimer contact
```

43. Ajouter la fonction deleteContact() dans le fichier Application\frontend\src\App.js

```
const deleteContactEvent = async () => {
  await dispatch(deleteContactAction(id))
  await dispatch(listerContacts());
  showDestroy(false);
}
```

44. Ajouter l'action deleteContactAction() dans le fichier Application\frontend\src\actions\contact.actions.js

```
export const deleteContactAction = (id) => {
  return async dispatch => {
    dispatch({ type: constantesactions.DELETE_CONTACT_REQUEST })
    try {
      console.log(id);
      const res = await
        axios.delete(`http://127.0.0.1:3000/contact/${id}/supprimer`)
      console.log('res' + res);
      if (res.status === 200) {
        console.log('data=' + id);
        dispatch({
          type: constantesactions.DELETE_CONTACT_SUCCESS,
          payload: { contactCreated: res.id }
        })
      }
    } catch (error) {
      console.log('erreur est : ' + error);
      console.log(error.response.id);
      dispatch({
        type: constantesactions.DELETE_CONTACT_FAILURE,
        payload: { error: error.response }
      })
    }
  }
}
```

Remarque :

Pour la ligne « `axios.get(`http://127.0.0.1:3000/contact/${id}/supprimer`)` »



45. Importer la méthode « deleteContactAction »

```
import { listerContacts, addContactAction, deleteContactAction } from
'./actions/contact.actions';
```

46. Modifier la méthode : handleCloseDestroy

```
const handleShowDestroy = (id) => {
  showDestroy(true);
  //alert("supprimer"+id);
  setId(id);
};
```

47. Ajouter la méthode suivante

```
const deleteContactEvent = async () => {
  await dispatch(deleteContactAction(id))
  await dispatch(listerContacts());
  showDestroy(false);
}
```

48. Maintenant, Ajouter le bouton « supprimer »,

```
<a href='#' className='btn btn-danger'
onClick={ (e) => handleShowDestroy(contact._id) } >supprimer</a>
```

Si j'aune une fonction paramétrée, ajouter la notation `= { () => deleteContact(contact._id) }` à votre fonction

49. Modifier dans le Modal de suppression

```
<Button variant="primary" onClick={deleteContactEvent}>
  Save Changes
</Button>
```

50. Interface pour modifier un contact

Changer la ligne qui déclencher le modal de la modification comme suit :

```
<a href='#' className='btn btn-success m-1' onClick={ () =>
handleShowEdit(contact._id) }>Modifier</a>
```

51. Modifier la fonction handleShowEdit () comme suit

```
const handleShowEdit = (id) => {
  showEdit(true);
  contacts.forEach(c => {
    if(c._id === id){
      console.log(c);
      setId(c._id);
      setNom(c.nom);
      setTel(c.tel);
    }
  });
};
```

52. Dans le modal de modification modifier le code de deux inputs (nom et tel) comme suit

```
.....
<Form.Control type="text"
value={nom} onChange={(e)=>setNom(e.target.value)} />
.....
<Form.Control type="tel" value={tel}
onChange={(e)=>setTel(e.target.value)} />
```

53. Ajouter les 3 contactes de la modification

```
EDIT_CONTACT_REQUEST: 'EDIT_CONTACT_REQUEST',
EDIT_CONTACT_SUCCESS: 'EDIT_CONTACT_SUCCESS',
EDIT_CONTACT_FAILURE: 'EDIT_CONTACT_FAILURE'
```

54. Ajouter l'action editContactAction

```

export const editContactAction = (id,data)=>{
  return async dispatch =>{
    dispatch({type:constantesactions.EDIT_CONTACT_REQUEST})
    try {
      //console.log(id,data);

      var params = new URLSearchParams();
      params.append('nom', data.nom);
      params.append('tel', data.tel);
      const res = await
    axios.put(`http://127.0.0.1:3000/contact/${id}/modifier`, params)
      //console.log('res'+res);
      if(res.status === 200){
        console.log('data='+id);
        dispatch({
          type:constantesactions.EDIT_CONTACT_SUCCESS,
          payload: {message: res.data}
        })
      }

    } catch (error) {
      console.log('erreur est : '+ error);
      console.log(error.response.data);
      //console.log(error.response.id);
      dispatch({
        type:constantesactions.EDIT_CONTACT_FAILURE,
        payload:{ error:error.response}
      })
    }
  }
}

```

55. Ajouter les reducers suivants

```

//edit contact
case constantesactions.EDIT_CONTACT_REQUEST:
  state ={
    ...state,
  }
  break;

case constantesactions.EDIT_CONTACT_SUCCESS:
  state ={
    ...state,
    createdC: action.payload.message
  }
  break;

case constantesactions.EDIT_CONTACT_FAILURE:
  state ={
    ...state,
    error: action.payload.error
  }
  break;
// fin edit contact

```

56. Modifier la fonction editContactEvent()

```
const editContactEvent= async ()=>{
  console.log('update for contact'+nom)
  const data ={
    nom,
    tel
  }
  dispatch(editContactAction(id,data))
  dispatch(listerContacts()) ;
  handleCloseEdit();
}
```

57. Importer la méthode editContactAction()

```
import { listerContacts, addContactAction, deleteContactAction , editContactAction}
from './actions/contact.actions';
```

58. Modifier le modal de la modification

```
<Button variant="primary" onClick={editContactEvent}>
  Save Changes
</Button>
```


Erreurs :

Erreur 1:

The href attribute requires a valid value to be accessible. Provide a valid, navigable address as the href value. If you cannot provide a valid href, but still need the element to resemble a link, use a button and change it with appropriate styles. Learn more: <https://github>.

Solution 1

Try to replace href="#" with href="/#" inside <a> element, this should fix the problem.

Erreur 2

axios Error: Request failed with status code 400

Solution 2

Ajouter ce code pour afficher le format de données envoyées (data)

```
console.log(error.response.data);
```

Source : <https://dev.to/zelig880/how-to-catch-the-body-of-an-axios-error-41k0>