

## Cours Inf3522 - Développement d'Applications N tiers

### Lab 12 : Mise en place du frontend pour notre service Web RESTful Spring Boot

Ce lab explique les étapes nécessaires pour démarrer le développement de la partie frontend. Tout d'abord, nous définirons les fonctionnalités que nous développons. Ensuite, nous réaliserons une maquette de l'interface utilisateur. Comme backend, nous utiliserons notre application Spring Boot du lab5. **Nous commencerons le développement en utilisant la version non sécurisée, lab4, du backend. Enfin, nous créerons l'application React que nous utiliserons dans notre développement frontend.**

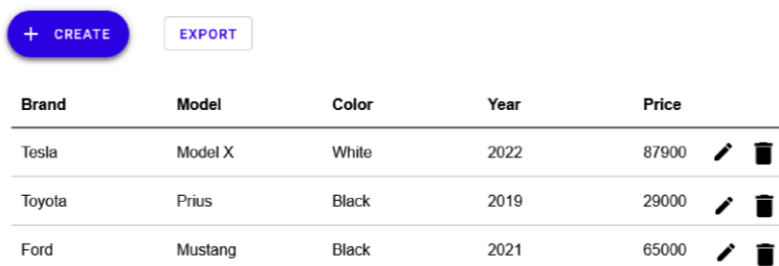
#### Pourquoi une maquette est nécessaire et comment procéder

Dans les premiers labs de ce cours, nous avons créé un backend qui fournit une API RESTful. Maintenant, il est temps de commencer à construire le frontend de notre application.

Nous allons créer un frontend qui répertorie les voitures de la base de données et qui propose la pagination, le tri et le filtrage. Il y a un bouton qui ouvre un formulaire modal pour ajouter de nouvelles voitures à la base de données. Sur chaque ligne du tableau des voitures, il y a un bouton pour supprimer ou modifier la voiture de la base de données. Le frontend contient également un lien ou un bouton pour exporter les données du tableau vers un fichier CSV.

Créons une maquette de notre interface utilisateur. Il existe de nombreuses applications différentes pour créer des maquettes, ou vous pouvez même utiliser un crayon et du papier. Vous pouvez également créer des maquettes interactives pour démontrer plusieurs fonctionnalités. Les modifications apportées à la maquette sont vraiment faciles et rapides à mettre en œuvre, par rapport aux modifications impliquant du code source réel du frontend.

La capture d'écran ci-dessous montre la maquette de notre frontend pour la liste des voitures :

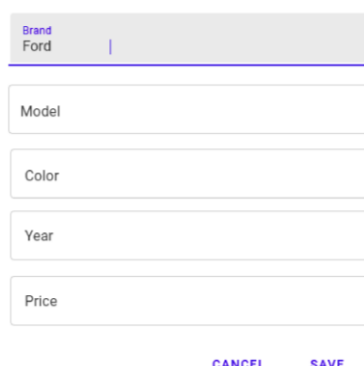


The screenshot shows a web application interface. At the top, there are two buttons: a purple button with a plus sign and the text '+ CREATE', and a light blue button with the text 'EXPORT'. Below these buttons is a table with five columns: 'Brand', 'Model', 'Color', 'Year', and 'Price'. The table contains three rows of data:

| Brand  | Model   | Color | Year | Price |
|--------|---------|-------|------|-------|
| Tesla  | Model X | White | 2022 | 87900 |
| Toyota | Prius   | Black | 2019 | 29000 |
| Ford   | Mustang | Black | 2021 | 65000 |

Each row in the table has two small icons at the end: a pencil icon for editing and a trash can icon for deleting.

Le formulaire modal qui s'ouvre lorsque l'utilisateur appuie sur le bouton Nouvelle voiture ressemble à ceci :



The screenshot shows a modal form for adding a new car. It has a header with the text 'Brand' and 'Ford' followed by a vertical line. Below the header are four input fields: 'Model', 'Color', 'Year', and 'Price'. At the bottom of the form are two buttons: 'CANCEL' and 'SAVE'.

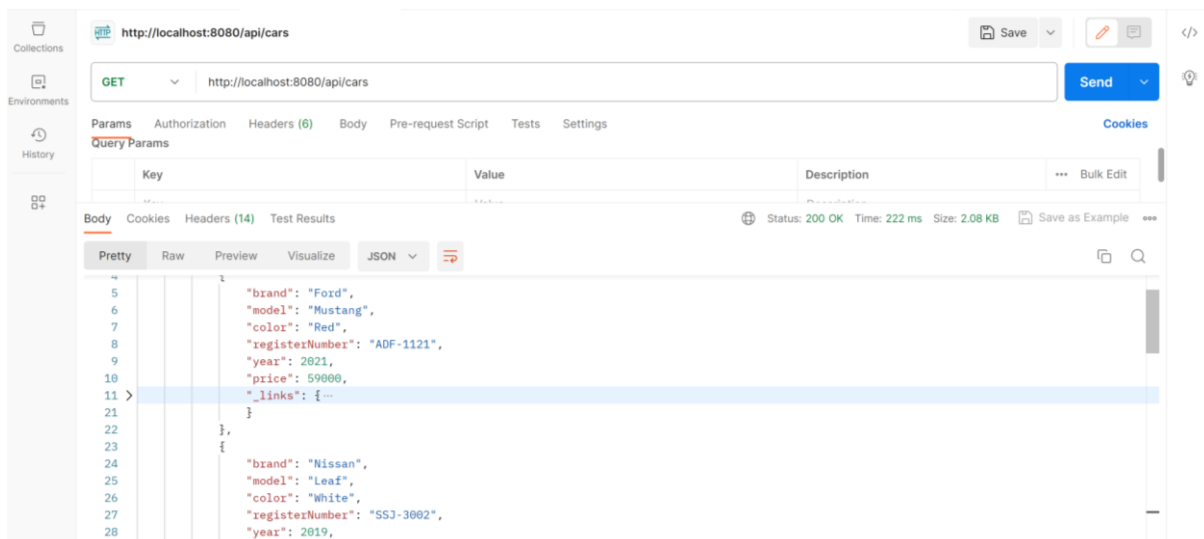
## Préparation de notre backend Spring Boot pour le développement frontend

Nous commençons le développement du frontend avec la version non sécurisée de notre backend. Dans la première phase, nous mettrons en œuvre toutes les fonctionnalités CRUD et nous testerons qu'elles fonctionnent correctement. Dans la deuxième phase, nous activerons la sécurité dans notre backend, apporterons les modifications nécessaires, et enfin, mettrons en place l'authentification.

Pour commencer, ouvrez l'application Spring Boot que nous avons créée dans le lab5. Ouvrez le fichier SecurityConfig.java qui définit la configuration de Spring Security. Temporairement, commentez la configuration actuelle et donnez l'accès à tout le monde pour tous les points d'accès (endpoints). Reportez-vous aux modifications suivantes :

```
SecurityConfig.java x
44
45 @Bean
46 SecurityFilterChain configureSecurity(HttpSecurity http) throws Exception {
47     // Add this row
48     http.csrf().disable().cors().and()
49     .authorizeHttpRequests().anyRequest().permitAll();
50
51     /*
52     http.csrf().disable().cors().and()
53     .sessionManagement()
54     .sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
55     .authorizeHttpRequests().requestMatchers(HttpMethod.POST, "/login").permitAll()
56     .anyRequest().authenticated().and()
57     .exceptionHandling().authenticationEntryPoint(exceptionHandler).and()
58     .addFilterBefore(authenticationFilter, UsernamePasswordAuthenticationFilter.class)
59
60     .httpBasic(withDefaults());
61     */
62     return http.build();
63 }
64
65
```

Maintenant, si vous démarrez la base de données MariaDB, exécutez le backend et envoyez la requête GET à l'endpoint **http://localhost:8080/api/cars** avec Postman, vous devriez obtenir toutes les voitures dans la réponse, comme le montre la capture d'écran suivante :



## Création de l'application React pour le frontend

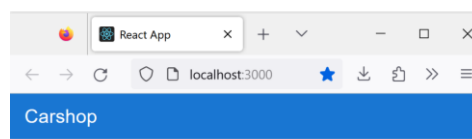
1. Créez une nouvelle application React carfront  
**npx create-react-app carfront**
2. Rentrez dans l'application et installez Material UI comme suit :  
**cd carfront**

**npm install @mui/material @emotion/react @emotion/styled**

3. Modifiez le composant App comme suit :

```
App.js
src > JS App.js > ...
1 import './App.css';
2 import AppBar from '@mui/material/AppBar';
3 import Toolbar from '@mui/material/Toolbar';
4 import Typography from '@mui/material/Typography';
5 function App() {
6   return (
7     <div className="App">
8       <AppBar position="static">
9         <Toolbar>
10           <Typography variant="h6">Carshop</Typography>
11         </Toolbar>
12       </AppBar>
13     </div>
14   );
15 }
16 export default App;
17
```

4. Enfin l'application **carfront** et vous obtenez :



## Ajout des fonctionnalités CRUD

Cette partie décrit comment nous pouvons mettre en œuvre les fonctionnalités CRUD (Create, Read, Update, Delete) dans notre interface frontend.

Nous allons utiliser les composants que nous avons vu dans le lab\_11 précédent. Nous allons récupérer des données depuis notre backend et présenter ces données dans un tableau. Ensuite, nous mettrons en place les fonctionnalités de suppression, d'édition et d'ajout. Dans la dernière section de cette partie, nous ajouterons des fonctionnalités pour pouvoir exporter les données vers un fichier CSV.

Au cours de cette partie, nous aborderons les sujets suivants :

### Création de la page de liste

Dans la première phase, nous allons créer la page de liste pour afficher les voitures avec des fonctionnalités de pagination, de filtrage et de tri. Exécutez votre backend Spring Boot non sécurisé. Les voitures peuvent être récupérées en envoyant la requête GET à l'URL <http://localhost:8080/api/cars>, comme illustré dans le lab\_4.

Maintenant, examinons les données JSON de la réponse. Le tableau de voitures se trouve dans le nœud **\_embedded.cars** des données de réponse JSON.

Une fois que nous savons comment récupérer les voitures depuis le backend, nous sommes prêts à mettre en œuvre la page de liste pour afficher les voitures. Les étapes suivantes décrivent cela en pratique :

1. Ouvrez l'application React **carfront** avec Visual Studio Code (l'application React précédemment).

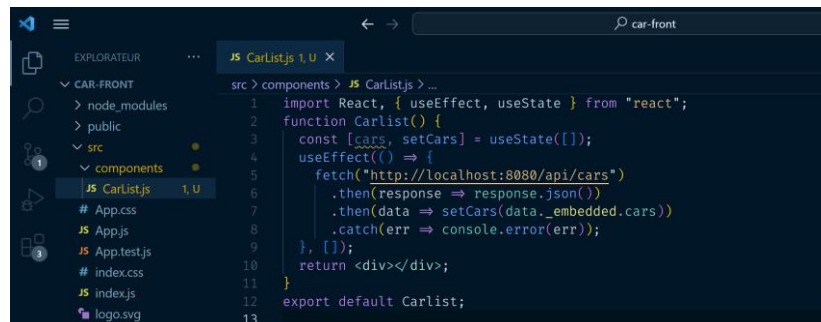
Lorsque l'application comporte plusieurs composants, il est recommandé de créer un dossier pour eux. Créez un nouveau dossier appelé **"components"** dans le dossier **"src"**.

Créez un nouveau fichier appelé **"Carlist.js"** dans le dossier **"components"**.

2. Ouvrez le fichier **"Carlist.js"** dans la vue de l'éditeur et modifiez le code de base du composant.

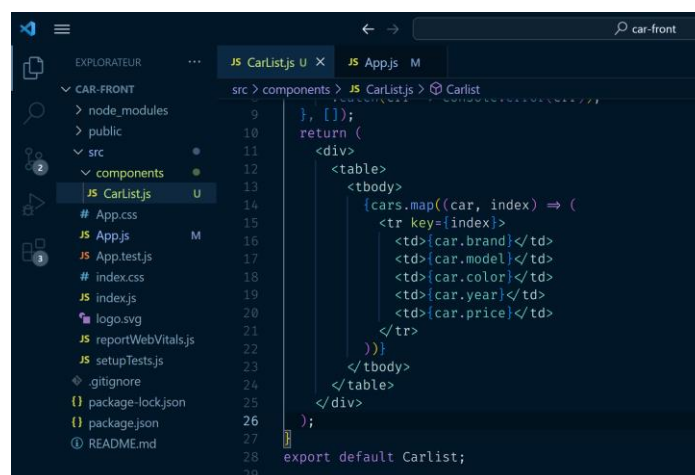
Nous avons besoin d'un état pour les voitures récupérées depuis l'API REST. Par conséquent, nous devons déclarer un état appelé **"cars"** et la valeur initiale est un tableau vide.

Exécutez la requête **"fetch"** dans le Hook **"useEffect"**. Nous passerons un tableau vide en tant que deuxième argument ; par conséquent, la requête **"fetch"** ne sera exécutée qu'une seule fois après le premier rendu. Les voitures extraites des données de réponse JSON seront enregistrées dans l'état appelé **"cars"** :



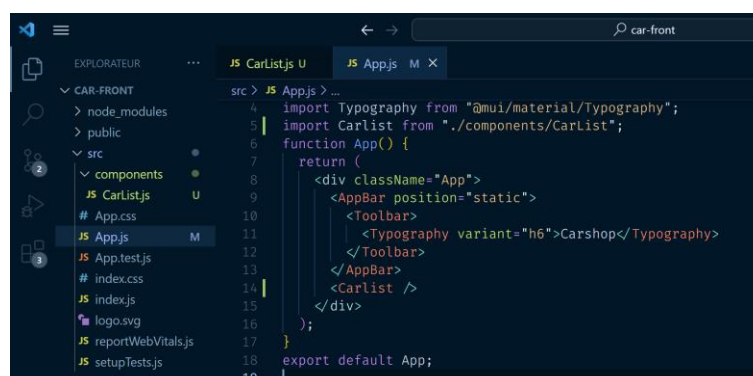
```
1 import React, { useEffect, useState } from "react";
2 function Carlist() {
3   const [cars, setCars] = useState([]);
4   useEffect(() => {
5     fetch("http://localhost:8080/api/cars")
6       .then(response => response.json())
7       .then(data => setCars(data._embedded.cars))
8       .catch(err => console.error(err));
9   }, []);
10   return <div></div>;
11 }
12 export default Carlist;
```

3. Utilisez la fonction **"map"** pour transformer les objets de voiture en lignes de tableau dans l'instruction **"return"** et ajoutez l'élément de tableau :



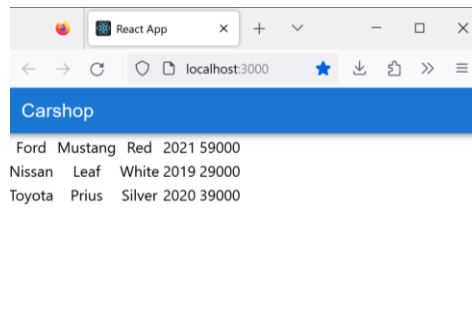
```
9   }, []);
10   return (
11     <div>
12       <table>
13         <tbody>
14           {cars.map((car, index) => (
15             <tr key={index}>
16               <td>{car.brand}</td>
17               <td>{car.model}</td>
18               <td>{car.color}</td>
19               <td>{car.year}</td>
20               <td>{car.price}</td>
21             </tr>
22           ))}
23         </tbody>
24       </table>
25     </div>
26   );
27 export default Carlist;
```

4. Enfin, nous devons importer et afficher le composant **"Carlist"** dans notre fichier **"App.js"**. Dans le fichier **"App.js"**, ajoutez l'instruction d'importation, puis ajoutez le composant **"Carlist"** (Ligne 14) à l'instruction **"return"** :



```
4 import Typography from "@mui/material/Typography";
5 import Carlist from "../components/Carlist";
6 function App() {
7   return (
8     <div className="App">
9       <AppBar position="static">
10         <Toolbar>
11           <Typography variant="h6">Carshop</Typography>
12         </Toolbar>
13       </AppBar>
14       <Carlist />
15     </div>
16   );
17 }
18 export default App;
```

5. Maintenant, si vous démarrez l'application React avec la commande "**npm start**", vous devriez voir la page de liste suivante. Notez que votre backend doit également être en cours d'exécution :



L'adresse URL du serveur peut se répéter plusieurs fois lorsque nous créons davantage de fonctionnalités CRUD, et elle changera lorsque le backend sera déployé sur un serveur autre que "**localhost**". Il est donc préférable de la définir comme une constante. Ensuite, lorsque la valeur de l'URL change, nous devons la modifier en un seul endroit.

Créons un nouveau fichier **constants.js** dans le dossier **src** de notre application :

1. Ouvrez le fichier dans l'éditeur et ajoutez la ligne suivante au fichier :

```
export const SERVER_URL = 'http://localhost:8080/';
```

2. Ensuite, nous importerons **SERVER\_URL** dans notre fichier **Carlist.js** et l'utiliserons dans la méthode **fetch** :

```
// Carlist.js
```

```
import { SERVER_URL } from '../constants.js'
```

```
// Utilisez la constante importée dans la méthode fetch
```

```
fetch(SERVER_URL + 'api/cars')
```

Nous avons déjà utilisé le composant **ag-grid** pour implémenter la grille de données, et il peut également être utilisé ici. Cependant, nous utiliserons le nouveau composant **MUI data grid** pour bénéficier des fonctionnalités de pagination, de filtrage et de tri prêtes à l'emploi :

1. Arrêtez le serveur de développement en appuyant sur **Ctrl + C** dans le terminal, puis installez la version communautaire du **MUI data grid**. Voici la commande d'installation actuelle, mais vous devriez vérifier la commande d'installation et l'utilisation les plus récentes dans la documentation de **MUI**. Après l'installation, redémarrez l'application :

```
npm install @mui/x-data-grid
```

2. Ensuite, importez le composant **DataGrid** dans votre fichier **Carlist.js** :

```
import { DataGrid } from '@mui/x-data-grid';
```

3. Nous devons également définir les colonnes de la grille de données, où "**field**" est la propriété de l'objet de la voiture. La prop "**headerName**" peut être utilisée pour définir le titre des colonnes. Nous définissons également la largeur des colonnes :

```
const columns = [  
  { field: 'brand', headerName: 'Marque', width: 200 },
```

```

    { field: 'model', headerName: 'Modèle', width: 200 },
    { field: 'color', headerName: 'Couleur', width: 200 },
    { field: 'year', headerName: 'Année', width: 150 },
    { field: 'price', headerName: 'Prix', width: 150 },
  ];

```

4. Ensuite, supprimez le tableau et tous ses éléments enfants de l'instruction **"return"** du composant, puis ajoutez le composant **DataGrid**. La source de données de la grille de données est l'état **"cars"**, qui contient les voitures extraites, et cela est défini à l'aide de la prop **"rows"**. Le composant de la grille de données exige que toutes les lignes aient une propriété ID unique définie à l'aide de la prop **"getRowId"**. Nous pouvons utiliser notre champ de lien de l'objet de voiture car il contient l'ID de voiture unique (**\_links.self.href**). Nous devons également définir la largeur et la hauteur de la grille dans l'élément div. Consultez le code source de l'instruction **"return"** suivante :

```

return (
  <div style={{ height: 500, width: '100%' }}>
    <DataGrid
      rows={cars}
      columns={columns}
      getRowId={row => row._links.self.href}
    />
  </div>
);

```

5. Avec le composant MUI data grid, nous avons obtenu toutes les fonctionnalités nécessaires (telles que le tri, le filtrage et la pagination) pour notre tableau avec une petite quantité de code. Maintenant, la page de liste ressemble à ce qui suit :

| Marque | Modèle  | Couleur | Année | Prix ↓ |
|--------|---------|---------|-------|--------|
| Ford   | Mustang | Red     | 2021  | 59000  |
| Toyota | Prius   | Silver  | 2020  | 39000  |
| Nissan | Leaf    | White   | 2019  | 29000  |

## Fonctionnalité de suppression

Les éléments peuvent être supprimés de la base de données en envoyant une requête de type DELETE à l'endpoint <http://localhost:8080/api/cars/{carId}>. Si nous examinons les données de réponse JSON, nous pouvons voir que chaque voiture contient un lien vers elle-même et qu'elle peut être accédée à partir du nœud **\_links.self.href**, comme indiqué dans la capture d'écran suivante. Nous avons déjà utilisé le champ « link » pour définir un identifiant unique pour chaque ligne dans la grille. Cet identifiant de ligne peut être utilisé dans la suppression, comme nous le verrons plus tard :

```

    "_links": {
      "self": {
        "href": "http://localhost:8080/api/cars/1"
      }
    },
  },
  ...

```

Les étapes suivantes démontrent comment implémenter la fonction de suppression :

1. Ici, nous allons créer un bouton pour chaque ligne dans le tableau. Le champ du bouton sera `_links.self.href`, qui est un lien vers une voiture. Si vous avez besoin d'un contenu de cellule plus complexe, vous pouvez utiliser une propriété `renderCell` que vous pouvez utiliser pour définir comment le contenu de la cellule est rendu. Ajoutons une nouvelle colonne au tableau en utilisant `renderCell` pour rendre l'élément du bouton. L'argument de ligne qui est passé à la fonction est un objet de ligne qui contient toutes les valeurs d'une ligne. Dans notre cas, il contient un lien vers une voiture dans chaque ligne, et cela est nécessaire pour la suppression. Le lien se trouve dans la propriété `id` de la ligne, et nous passerons cette valeur à une fonction de suppression. Référez-vous au code source suivant. Nous ne voulons pas activer le tri et le filtrage pour la colonne des boutons, par conséquent, les propriétés `filterable` et `sortable` sont définies sur `false`. Le bouton invoque la fonction `onDelClick` lorsque vous appuyez dessus et passe un lien (`row.id`) à la fonction en tant qu'argument :

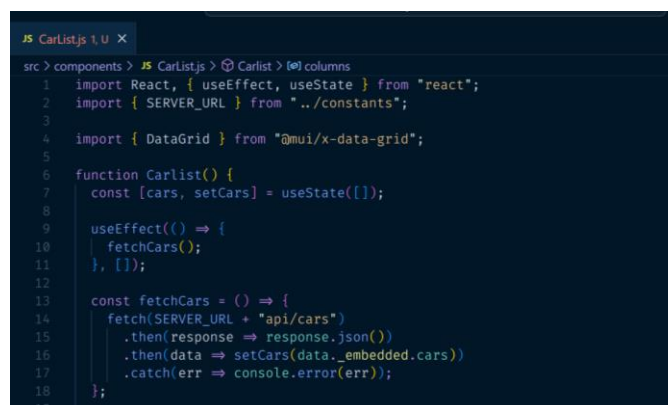


```

14
15 const columns = [
16   { field: "brand", headerName: "Marque", width: 200 },
17   { field: "model", headerName: "Modèle", width: 200 },
18   { field: "color", headerName: "Couleur", width: 200 },
19   { field: "year", headerName: "Année", width: 150 },
20   { field: "price", headerName: "Prix", width: 150 },
21   {
22     field: "_links.self.href",
23     headerName: "",
24     sortable: false,
25     filterable: false,
26     renderCell: row => (
27       <button onClick={() => onDelClick(row.id)}>Delete</button>
28     ),
29   },
30 ];

```

2. Ensuite, nous implémentons la fonction `onDelClick`. Mais d'abord, retirons la méthode `fetch` du Hook `useEffect`. C'est nécessaire car nous voulons également appeler `fetch` après la suppression de la voiture pour montrer à l'utilisateur une liste mise à jour des voitures. Créez une nouvelle fonction appelée `fetchCars`, et copiez le code du Hook `useEffect` dans une nouvelle fonction. Ensuite, appelez la fonction `fetchCars` depuis le Hook `useEffect` pour récupérer les voitures :



```

1  import React, { useEffect, useState } from "react";
2  import { SERVER_URL } from "../constants";
3
4  import { DataGrid } from "@mui/x-data-grid";
5
6  function Carlist() {
7    const [cars, setCars] = useState([]);
8
9    useEffect(() => {
10      fetchCars();
11    }, []);
12
13    const fetchCars = () => {
14      fetch(SERVER_URL + "api/cars")
15        .then(response => response.json())
16        .then(data => setCars(data._embedded.cars))
17        .catch(err => console.error(err));
18    };
19
20

```

3. Implémentez la fonction `onDelClick`. Nous envoyons la requête `DELETE` à un lien de voiture, et lorsque la requête `DELETE` réussit, nous rafraîchissons la page de liste en appelant la fonction `fetchCars` (Lignes 20 – 24) :

```
src > components > JS CarListjs > Carlist
3
4 import { DataGrid } from "@mui/x-data-grid";
5
6 function Carlist() {
7   const [cars, setCars] = useState([]);
8
9   useEffect(() => {
10     fetchCars();
11   }, []);
12
13   const fetchCars = () => {
14     fetch(SERVER_URL + "api/cars")
15       .then(response => response.json())
16       .then(data => setCars(data._embedded.cars))
17       .catch(err => console.error(err));
18   };
19
20   const onDelClick = url => {
21     fetch(url, { method: "DELETE" })
22       .then(response => fetchCars())
23       .catch(err => console.error(err));
24   };
25 }
```

Lorsque vous démarrez votre application, l'interface utilisateur devrait ressembler à la capture d'écran suivante. La voiture disparaît de la liste lorsque le bouton Supprimer est pressé. Notez qu'après des suppressions, vous pouvez redémarrer le backend pour réinitialiser la base de données :

| Marque | Modèle  | Couleur | Année | Prix  |        |
|--------|---------|---------|-------|-------|--------|
| Ford   | Mustang | Red     | 2021  | 59000 | Delete |
| Nissan | Leaf    | White   | 2019  | 29000 | Delete |
| Toyota | Prius   | Silver  | 2020  | 39000 | Delete |

Rows per page: 100 1-3 of 3

Vous pouvez également remarquer que lorsque vous cliquez sur n'importe quelle ligne dans la grille, la ligne est sélectionnée. Vous pouvez désactiver cela en définissant la propriété `disableSelectionOnClick` de la grille sur `true` : Ligne 48.

```
43 return (
44   <div style={{ height: 500, width: "100%" }}>
45     <DataGrid
46       rows={cars}
47       columns={columns}
48       disableSelectionOnClick={true}
49       getRowId={row => row._links.self.href}
50     />
51   </div>
52 );
53
54 export default Carlist;
```



Ce serait bien de montrer à l'utilisateur des retours d'informations en cas de suppression réussie ou s'il y a des erreurs.

4. Mettre en place un message toast pour afficher l'état de la suppression. Pour cela, nous allons utiliser le composant Snackbar de MUI. Nous devons importer le composant Snackbar en ajoutant la déclaration d'importation suivante dans votre fichier Carlist.js :

```
import Snackbar from '@mui/material/Snackbar';
```

La propriété open du composant Snackbar est une valeur booléenne, et si elle est vraie, le composant est affiché. Déclarons donc un état appelé "open" pour gérer la visibilité de notre composant Snackbar. La valeur initiale est "false" car le message n'est affiché qu'après la suppression :

```
// Carlist.js
```

```
const [open, setOpen] = useState(false);
```

5. Ensuite, nous ajoutons le composant Snackbar dans l'instruction return après le composant MUI Data Grid. La propriété autoHideDuration définit le temps en millisecondes où la fonction onClose est appelée automatiquement, et le message disparaît. La propriété message définit le message à afficher :

```
46   return (
47     <div style={{ height: 500, width: "100%" }}>
48       <DataGrid
49         rows={cars}
50         columns={columns}
51         disableSelectionOnClick={true}
52         getRowId={row => row._links.self.href}
53       />
54       <Snackbar
55         open={open}
56         autoHideDuration={2000}
57         onClose={() => setOpen(false)}
58         message="Voiture supprimée"
59       />
60     </div>
61   );
62 }
```

6. Finalement, nous définissons l'état "open" sur "true" après la suppression, et le message toast s'affiche

```
23   const onDelClick = url => {
24     fetch(url, { method: "DELETE" })
25       .then(response => {
26         fetchCars();
27         setOpen(true);
28       })
29       .catch(err => console.error(err));
30   };
31 }
```

Maintenant, vous verrez le message toast lorsque la voiture est supprimée, comme indiqué dans la capture d'écran suivante :



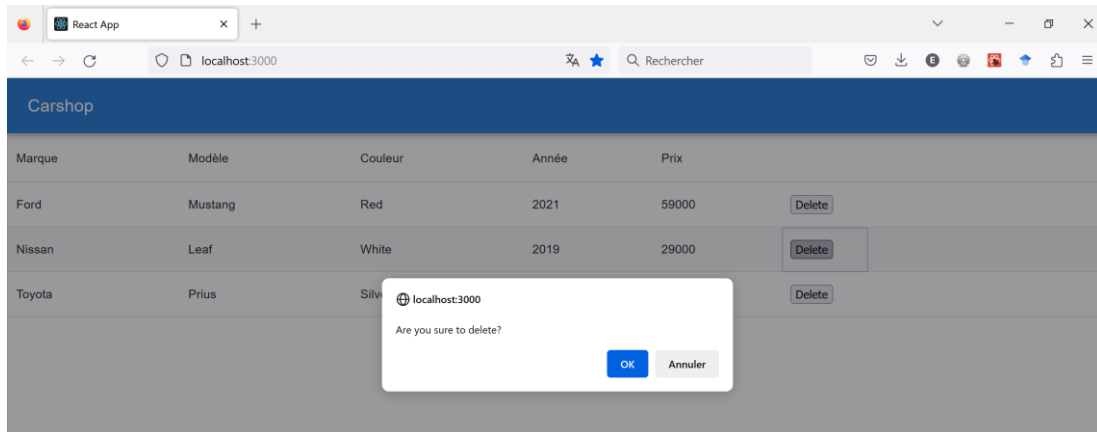
7. Pour éviter la suppression accidentelle de la voiture, il serait judicieux d'avoir une boîte de dialogue de confirmation après avoir appuyé sur le bouton Supprimer. Nous allons implémenter cela en utilisant la méthode confirm de l'objet window. Ajoutez confirm à la méthode onDelClick (Ligne 24) :

```

33   const onDeleteClick = url => {
34     if (window.confirm("Are you sure to delete?")) {
35       fetch(url, { method: "DELETE" })
36         .then(response => {
37           fetchCars();
38           setOpen(true);
39         })
40         .catch(err => console.error(err));
41     }
42   };
43 }

```

Si vous appuyez sur le bouton Supprimer maintenant, la boîte de dialogue de confirmation s'ouvrira, et la voiture ne sera supprimée que si vous appuyez sur le bouton OK.



Enfin, nous vérifierons également le statut de la réponse pour nous assurer que la suppression s'est bien déroulée. Comme nous l'avons déjà appris, l'objet réponse a la propriété "ok" que nous pouvons utiliser pour vérifier si la réponse a été réussie :

```

23   const onDeleteClick = url => {
24     if (window.confirm("Êtes-vous sûr de vouloir supprimer ?")) {
25       fetch(url, { method: "DELETE" })
26         .then(response => {
27           if (response.ok) {
28             fetchCars();
29             setOpen(true);
30           } else {
31             alert("Quelque chose s'est mal passé !");
32           }
33         })
34         .catch(err => console.error(err));
35     }
36   };
37 }

```

Ensuite, nous commencerons la mise en œuvre de la fonctionnalité pour ajouter une nouvelle voiture.

### Fonctionnalité d'ajout

La prochaine étape consiste à créer une fonction d'ajout pour l'interface utilisateur. Nous allons implémenter cela en utilisant la boîte de dialogue modale MUI. Nous avons déjà examiné l'utilisation du formulaire modal MUI dans le lab11, « Composants tiers utiles pour React ». Nous allons ajouter le bouton "Nouvelle voiture" à l'interface utilisateur, qui ouvre le formulaire modal lorsque vous appuyez dessus. Le formulaire modal contient tous les champs nécessaires pour ajouter une nouvelle voiture, ainsi que les boutons de sauvegarde et d'annulation.

Nous avons déjà installé la bibliothèque de composants MUI dans notre application frontend au début de ce lab.

Les étapes suivantes vous montrent comment créer la fonction d'ajout en utilisant le composant de boîte de dialogue modale :

Créez un nouveau fichier appelé AddCar.js dans le dossier des composants et écrivez du code de base pour un composant fonction, comme indiqué ici. Ajoutez les importations pour les composants MUI Dialog :

```
JS AddCar.js 4, U X
src > components > JS AddCar.js > ...
1 import React from "react";
2 import Dialog from "@mui/material/Dialog";
3 import DialogActions from "@mui/material/DialogActions";
4 import DialogContent from "@mui/material/DialogContent";
5 import DialogTitle from "@mui/material/DialogTitle";
6
7 function AddCar(props) {
8   return <div></div>;
9 }
10
11 export default AddCar;
12
```

(À noter que le contenu du composant AddCar sera développé davantage pour inclure le formulaire et la logique de sauvegarde des nouvelles voitures.)

Déclarez un état qui contient tous les champs de la voiture en utilisant le Hook useState. Pour la boîte de dialogue, nous avons également besoin d'un état booléen pour définir la visibilité du formulaire de la boîte de dialogue :

```
JS AddCar.js 8, U X
src > components > JS AddCar.js > ...
1 import React, { useState } from "react";
2 import Dialog from "@mui/material/Dialog";
3 import DialogActions from "@mui/material/DialogActions";
4 import DialogContent from "@mui/material/DialogContent";
5 import DialogTitle from "@mui/material/DialogTitle";
6
7 function AddCar(props) {
8   const [open, setOpen] = useState(false);
9   const [car, setCar] = useState({
10     brand: "",
11     model: "",
12     color: "",
13     year: "",
14     fuel: "",
15     price: "",
16   });
17   return <div></div>;
18 }
19
20
21 export default AddCar;
22
```

Ensuite, nous ajoutons deux fonctions pour fermer et ouvrir le formulaire de la boîte de dialogue. Les fonctions handleClickOpen et handleClose définissent la valeur de l'état "open", ce qui affecte la visibilité du formulaire modal :

```
JS AddCar.js U X
src > components > JS AddCar.js > AddCar
1 import React, { useState } from "react";
2 import Dialog from "@mui/material/Dialog";
3 import DialogActions from "@mui/material/DialogActions";
4 import DialogContent from "@mui/material/DialogContent";
5 import DialogTitle from "@mui/material/DialogTitle";
6
7 function AddCar(props) {
8   const [open, setOpen] = useState(false);
9   const [car, setCar] = useState({
10     brand: "",
11     model: "",
12     color: "",
13     year: "",
14     fuel: "",
15     price: "",
16   });
17
18   const handleClickOpen = () => {
19     setOpen(true);
20   };
21
22   const handleClose = () => {
23     setOpen(false);
24   };
25 }
```

Ces fonctions seront utilisées pour gérer l'ouverture et la fermeture de la boîte de dialogue du formulaire d'ajout.

Ajoutez un composant Dialog à l'intérieur de l'instruction return du composant AddCar. Le formulaire contient le composant Dialog MUI avec des boutons et les champs de saisie nécessaires pour collecter les données de la voiture. Le bouton qui ouvre la fenêtre modale, qui sera affichée sur la page de la liste de voitures, doit être à l'extérieur du composant Dialog. Tous les champs de saisie doivent avoir l'attribut "name" avec une valeur identique au nom de l'état dans lequel la valeur sera enregistrée. Les champs de saisie ont également la propriété "onChange", qui enregistre la valeur dans l'état en invoquant la fonction "handleChange" :

```
JS AddCar.js U X
src > components > JS AddCar.js > AddCar > handleClickOpen
29
30   return (
31     <div>
32       <button onClick={handleClickOpen}>New Car</button>
33       <Dialog open={open} onClose={handleClose}>
34         <DialogTitle>New car</DialogTitle>
35         <DialogContent>
36           <input
37             placeholder="Brand"
38             name="brand"
39             value={car.brand}
40             onChange={handleChange}
41           />
42           <br />
43           <input
44             placeholder="Model"
45             name="model"
46             value={car.model}
47             onChange={handleChange}
48           />
49           <br />
50           <input
51             placeholder="Color"
52             name="color"
53             value={car.color}
54             onChange={handleChange}
55           />
56           <br />
57           <input
58             placeholder="Year"
```

```
JS AddCar.js U X
src > components > JS AddCar.js > AddCar > handleClickOpen
59
60           <input
61             placeholder="Year"
62             name="year"
63             value={car.year}
64             onChange={handleChange}
65           />
66           <br />
67           <input
68             placeholder="Price"
69             name="price"
70             value={car.price}
71             onChange={handleChange}
72           />
73         </DialogContent>
74         <DialogActions>
75           <button onClick={handleClose}>Cancel</button>
76           <button onClick={handleClose}>Save</button>
77         </DialogActions>
78       </Dialog>
79     </div>
80   );
81
82   export default AddCar;
```

Mettez en œuvre la fonction "addCar" dans le fichier "Carlist.js", qui enverra la requête POST à l'endpoint backend "api/cars". La requête inclura le nouvel objet de voiture dans le corps (body) et l'en-tête 'Content-Type': 'application/json'. L'en-tête est nécessaire car l'objet de la voiture est converti en une chaîne JSON à l'aide de la méthode "JSON.stringify()", (ligne 23 - 37) :

```

17 fetch(SERVER_URL + "api/cars")
18   .then(response => response.json())
19   .then(data => setCars(data._embedded.cars))
20   .catch(err => console.error(err));
21 };
22
23 const addCar = car => {
24   fetch(SERVER_URL + "api/cars", {
25     method: "POST",
26     headers: { "Content-Type": "application/json" },
27     body: JSON.stringify(car),
28   })
29   .then(response => {
30     if (response.ok) {
31       fetchCars();
32     } else {
33       alert("Something went wrong!");
34     }
35   })
36   .catch(err => console.error(err));
37 };

```

Importez le composant AddCar dans le fichier Carlist.js :

**import AddCar from './AddCar.js';**

Ajoutez le composant AddCar à l'instruction "return" du fichier Carlist.js et transmettez la fonction "addCar" en tant que propriété (props) au composant AddCar. Cela nous permet d'appeler cette fonction à partir du composant AddCar. Maintenant, l'instruction "return" du fichier Carlist.js devrait ressembler à ceci :

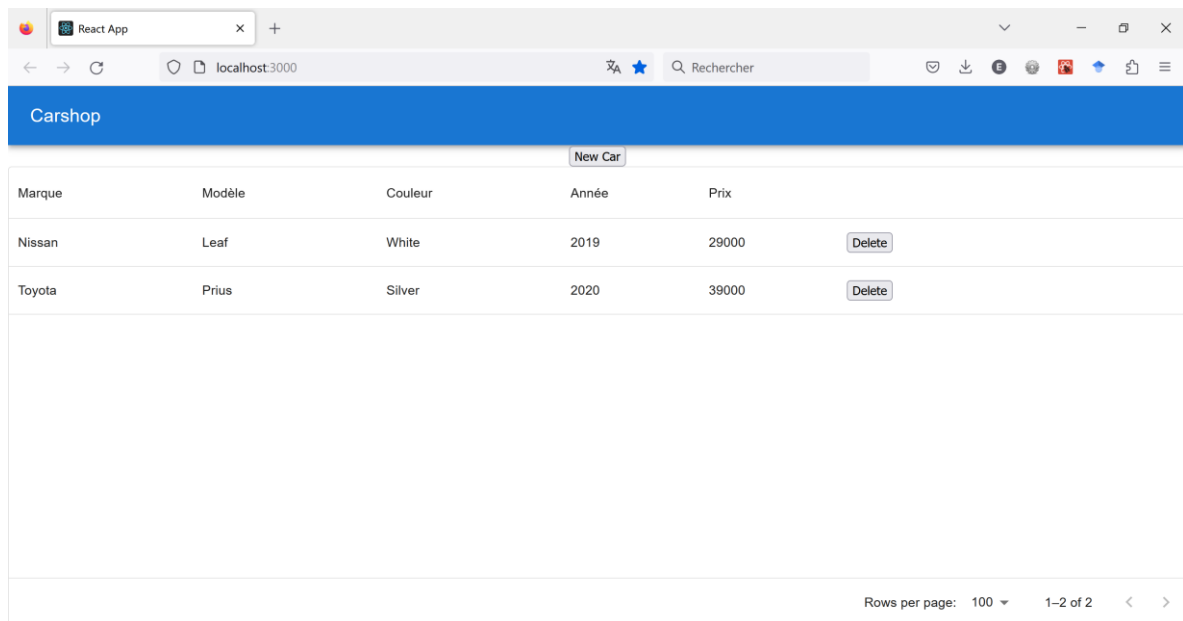
```

69   },
70   },
71 ];
72
73 return (
74   <React.Fragment>
75     <AddCar addCar={addCar} />
76     <div style={{ height: 500, width: "100%" }}>
77       <DataGrid
78         rows={cars}
79         columns={columns}
80         disableSelectionOnClick={true}
81         getRowId={row => row._links.self.href}
82       />
83       <Snackbar
84         open={open}
85         autoHideDuration={2000}
86         onClose={() => setOpen(false)}
87         message="Voiture supprimée"
88       />
89     </div>
90   </React.Fragment>
91 );
92 }
93 export default Carlist;
94

```

Cela ajoute le composant AddCar à la page de liste de voitures et lui permet d'accéder à la fonction "addCar" pour gérer l'ajout de nouvelles voitures.

Si vous démarrez l'application Carshop, elle devrait maintenant ressembler à ce qui suit, et si vous appuyez sur le bouton "Nouvelle voiture", la boîte de dialogue modale devrait s'ouvrir :



Créez une fonction appelée "handleSave" dans le fichier "AddCar.js" (Ligne 26 – 29). La fonction "handleSave" appelle la fonction "addCar", qui peut être accédée via les "props", et transmet l'objet d'état de la voiture à cette fonction. Enfin, le formulaire modal est fermé, et la liste des voitures est mise à jour :

```

10   brand: "",
11   model: "",
12   color: "",
13   year: "",
14   fuel: "",
15   price: "",
16   });
17
18   const handleClickOpen = () => {
19     setOpen(true);
20   };
21
22   const handleClose = () => {
23     setOpen(false);
24   };
25
26   const handleSave = () => {
27     props.addCar(car);
28     handleClose();
29   };
30

```

Ensuite, vous devez modifier le bouton "Enregistrer" du composant "AddCar" pour appeler la fonction "handleSave" lorsqu'il est cliqué :

```

70     placeholder="Price"
71     name="price"
72     value={car.price}
73     onChange={handleChange}
74   </br />
75   </DialogContent>
76   <DialogActions>
77     <button onClick={handleClose}>Annuler</button>
78     <button onClick={handleSave}>Enregistrer</button>
79   </DialogActions>
80 </Dialog>
81 </div>
82 );
83 }
84 }
85
86 export default AddCar;
87

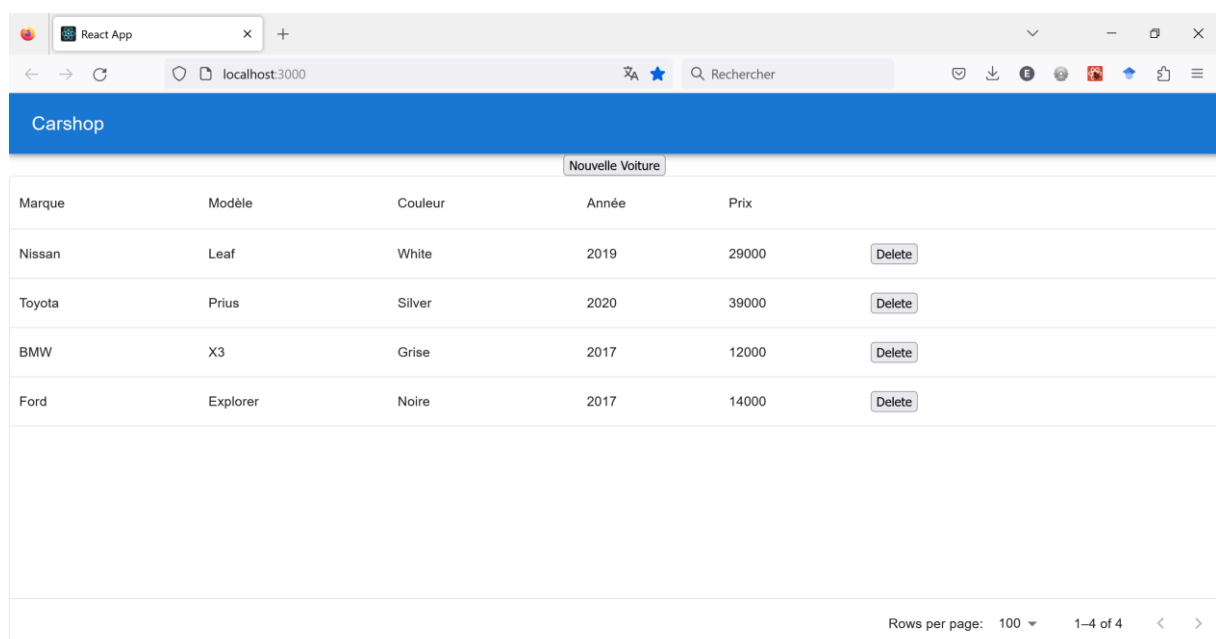
```

Maintenant, vous pouvez ouvrir le formulaire modal en appuyant sur le bouton "Nouvelle voiture". Ensuite, vous pouvez remplir le formulaire avec des données et appuyer sur le bouton "Enregistrer". À ce stade, le formulaire n'a pas un aspect agréable, mais nous allons le styliser dans le lab\_13.

### New car

|          |
|----------|
| Ford     |
| Explorer |
| Noire    |
| 2017     |
| 14000    |

Après l'enregistrement, la page de liste est rafraîchie, et la nouvelle voiture peut être vue dans la liste:



| Marque | Modèle   | Couleur | Année | Prix  |                                       |
|--------|----------|---------|-------|-------|---------------------------------------|
| Nissan | Leaf     | White   | 2019  | 29000 | <input type="button" value="Delete"/> |
| Toyota | Prius    | Silver  | 2020  | 39000 | <input type="button" value="Delete"/> |
| BMW    | X3       | Grise   | 2017  | 12000 | <input type="button" value="Delete"/> |
| Ford   | Explorer | Noire   | 2017  | 14000 | <input type="button" value="Delete"/> |

Rows per page: 100 1-4 of 4

Ensuite, nous commencerons à mettre en œuvre la fonctionnalité de modification en ce qui concerne notre interface frontend.

### Fonctionnalité de modification

Nous allons implémenter la fonctionnalité de modification en ajoutant le bouton "Modifier" à chaque ligne du tableau.

Lorsque le bouton "Modifier" de la ligne est pressé, il ouvre le formulaire modal, où l'utilisateur peut modifier la voiture existante et enregistrer enfin les changements :

Tout d'abord, nous allons créer une ébauche du composant "EditCar", qui sera le formulaire pour modifier une voiture existante. Créez un nouveau fichier appelé "EditCar.js" dans le dossier des composants. Le code du composant "EditCar" est similaire à celui du composant "AddCar", mais pour l'instant, dans la fonction "handleSave", nous devrions appeler la fonction "update" que nous mettrons en œuvre ultérieurement :

```

JS EditCar.js 9, U
src > components > JS EditCar.js > ...
1 import React, { useState } from "react";
2 import Dialog from "@mui/material/Dialog";
3 import DialogActions from "@mui/material/DialogActions";
4 import DialogContent from "@mui/material/DialogContent";
5 import DialogTitle from "@mui/material/DialogTitle";
6 function EditCar(props) {
7   const [open, setOpen] = useState(false);
8   const [car, setCar] = useState({
9     brand: "",
10    model: "",
11    color: "",
12    year: "",
13    price: "",
14  });
15   // Open the modal form
16   const handleClickOpen = () => {
17     setOpen(true);
18   };
19   // Close the modal form
20   const handleClose = () => {
21     setOpen(false);
22   };
23   const handleChange = event => {
24     setCar({ ...car, [event.target.name]: event.target.value });
25   };
26   // Update car and close modal form
27   const handleSave = () => {};
28   return <div></div>;
29 }
30 export default EditCar;

```

Ensuite, nous rendons le formulaire de dialogue de modification dans l'instruction "return" :

```

JS EditCar.js U X
src > components > JS EditCar.js > EditCar
26
27 const handleSave = () => {};
28 return (
29   <div>
30     <button onClick={handleClickOpen}>Edit</button>
31     <Dialog open={open} onClose={handleClose}>
32       <DialogTitle>Edit car</DialogTitle>
33       <DialogContent>
34         <input
35           placeholder="Brand"
36           name="brand"
37           value={car.brand}
38           onChange={handleChange}
39         />
40         <br />
41         <input
42           placeholder="Model"
43           name="model"
44           value={car.model}
45           onChange={handleChange}
46         />
47         <br />
48         <input
49           placeholder="Color"
50           name="color"
51           value={car.color}
52           onChange={handleChange}
53         />
54         <br />

```

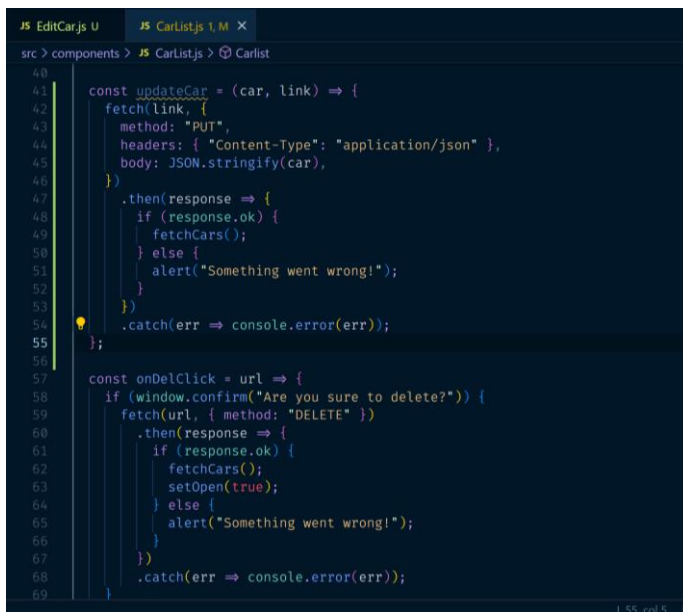
```

JS EditCar.js U X
src > components > JS EditCar.js > EditCar
53         />
54         <br />
55         <input
56           placeholder="Year"
57           name="year"
58           value={car.year}
59           onChange={handleChange}
60         />
61         <br />
62         <input
63           placeholder="Price"
64           name="price"
65           value={car.price}
66           onChange={handleChange}
67         />
68         <br />
69       </DialogContent>
70       <DialogActions>
71         <button onClick={handleClose}>Cancel</button>
72         <button onClick={handleSave}>Save</button>
73       </DialogActions>
74     </Dialog>
75   </div>
76 );
77
78 export default EditCar;
79

```



Pour mettre à jour les données de la voiture, nous devons envoyer une requête PUT à l'URL `http://localhost:8080/api/cars/[carid]`. Le lien sera le même que celui utilisé pour la fonction de suppression. La requête contient l'objet de la voiture mise à jour dans le corps (body) et l'en-tête 'Content-Type': 'application/json' que nous avons dans la fonction d'ajout. Créez une nouvelle fonction appelée "updateCar" dans le fichier "Carlist.js". Le code source de la fonction est présent dans l'extrait de code suivant. La fonction prend deux arguments : l'objet de la voiture mis à jour et l'URL de la requête. Après une mise à jour réussie, nous récupérerons les voitures et mettrons à jour la liste (Ligne 41 – 55) :



```
40
41 const updateCar = (car, link) => {
42   fetch(link, {
43     method: "PUT",
44     headers: { "Content-Type": "application/json" },
45     body: JSON.stringify(car),
46   })
47   .then(response => {
48     if (response.ok) {
49       fetchCars();
50     } else {
51       alert("Something went wrong!");
52     }
53   })
54   .catch(err => console.error(err));
55 };
56
57 const onDeleteClick = url => {
58   if (window.confirm("Are you sure to delete?")) {
59     fetch(url, { method: "DELETE" })
60     .then(response => {
61       if (response.ok) {
62         fetchCars();
63         setOpen(true);
64       } else {
65         alert("Something went wrong!");
66       }
67     })
68     .catch(err => console.error(err));
69   }
```

Ensuite, nous allons importer le composant "EditCar" dans le composant "Carlist" pour pouvoir l'afficher dans la liste de voitures. Ajoutez la déclaration d'importation suivante dans le fichier "Carlist.js" :

```
import EditCar from './EditCar.js';
```

Maintenant, ajoutez le composant "EditCar" aux colonnes du tableau de la même manière que nous l'avons fait pour la fonction de suppression. Le composant "EditCar" est rendu dans les cellules du tableau, et il affiche uniquement le bouton "Modifier". Cela est dû au fait que le formulaire modal n'est pas visible avant que le bouton ne soit pressé. Lorsque l'utilisateur appuie sur le bouton "Modifier", il met à jour la valeur de l'état "open" à true dans le composant "EditCar", et le formulaire modal s'affiche. Nous transmettons deux "props" au composant "EditCar". La première "prop" est "row", qui sera l'objet de la ligne contenant le lien et les données de la voiture dont nous avons besoin pour la mise à jour. La deuxième est la fonction "updateCar", que nous devons appeler depuis le composant "EditCar" pour pouvoir enregistrer les modifications (Ligne 81 – 86) :

```

JS EditCar.js U JS CarList.js M X
src > components > JS CarList.js > ...
73
74 const columns = [
75   { field: "brand", headerName: "Marque", width: 200 },
76   { field: "model", headerName: "Modèle", width: 200 },
77   { field: "color", headerName: "Couleur", width: 200 },
78   { field: "year", headerName: "Année", width: 150 },
79   { field: "price", headerName: "Prix", width: 150 },
80   {
81     field: "_links.car.href",
82     headerName: "",
83     sortable: false,
84     filterable: false,
85     renderCell: row => <EditCar data={row} updateCar={updateCar} />,
86   },
87   {
88     field: "_links.self.href",
89     headerName: "",
90     sortable: false,
91     filterable: false,
92     renderCell: row => (
93       <button onClick={() => onDeleteClick(row.id)}>Delete</button>
94     ),
95   },
96 ];
97

```

Nous avons maintenant ajouté le bouton "Modifier la voiture" à la grille de données, et nous pouvons ouvrir le formulaire de modification. Notre objectif est de remplir les champs du formulaire de modification en utilisant les données de la ligne où le bouton "Modifier" est pressé.

Ensuite, nous effectuerons les modifications finales dans le fichier "EditCar.js". Nous obtenons la voiture à éditer à partir de la "prop" "data", que nous utilisons pour remplir le formulaire avec les valeurs existantes de la voiture. Modifiez la fonction "handleClickOpen" dans le fichier "EditCar.js". Maintenant, lorsque le formulaire est ouvert, l'état de la voiture est mis à jour avec les valeurs de la "prop" "data". La "prop" "data" contient une propriété "row" qui contient l'objet de la voiture (Ligne 16 – 26):

```

JS EditCar.js U X JS CarList.js M
src > components > JS EditCar.js > EditCar
9
10 brand: "",
11 model: "",
12 color: "",
13 year: "",
14 price: "",
15 });
16 // Open the modal form
17 const handleClickOpen = () => {
18   setCar({
19     brand: props.data.row.brand,
20     model: props.data.row.model,
21     color: props.data.row.color,
22     year: props.data.row.year,
23     fuel: props.data.row.fuel,
24     price: props.data.row.price,
25   });
26   setOpen(true);
27 };
28 // Close the modal form
29 const handleClose = () => {
30   setOpen(false);
31 };
32 const handleChange = event => {
33   setCar({ ...car, [event.target.name]: event.target.value });
34 };
35

```

Enfin, nous allons modifier la fonction "handleSave" dans le fichier "EditCar.js" et appeler la fonction "updateCar" en utilisant les "props". Le premier argument est l'état de la voiture qui contient l'objet de la voiture mis à jour. Le deuxième argument est la propriété "id" de la "prop" "data", qui est le lien vers une voiture (Ligne 35 – 38):

```

27 // Close the modal form
28 const handleClose = () => {
29   setOpen(false);
30 };
31 const handleChange = event => {
32   setCar({ ...car, [event.target.name]: event.target.value });
33 };
34
35 const handleSave = () => {
36   props.updateCar(car, props.data.id);
37   handleClose();
38 };
39

```

Si vous appuyez sur le bouton "Modifier" dans le tableau, cela ouvre le formulaire de modification modal et affiche la voiture de cette ligne. Les valeurs mises à jour sont enregistrées dans la base de données lorsque vous appuyez sur le bouton "Enregistrer". À présent, nous avons mis en œuvre toutes les fonctionnalités CRUD en ce qui concerne notre interface frontend.

