

Cours Inf3522 - Développement d'Applications N tiers

Lab 9 : API Context, Événements, Listes et formulaires

Ce lab présente l'utilisation de l'API contexte, la gestion des événements, des listes et des formulaires.

L'API « Context »

Passer des données à l'aide de props peut être fastidieux si votre arborescence de composants est profonde et complexe.

Vous devez passer les données à travers tous les composants de l'arborescence des composants. L'API « context » résout ce problème et il est recommandé de l'utiliser pour les données globales dont vous pourriez avoir besoin dans plusieurs composants de votre arborescence de composants, par exemple un thème ou un utilisateur authentifié.

Le contexte est créé en utilisant la méthode **createContext** et elle prend un argument qui définit la valeur par défaut. Vous pouvez créer votre propre fichier pour le contexte, et le code ressemble à :

```
import React from 'react';
const AuthContext = React.createContext("");
export default AuthContext;
```

Ensuite, nous utiliserons un composant fournisseur de contexte qui rend notre contexte disponible pour les autres composants. Le composant fournisseur de contexte a une props **value** qui sera transmise aux composants consommateurs. Dans l'exemple suivant, nous avons enveloppé **<MyComponent />** dans le composant fournisseur de contexte, de ce fait, la valeur **userName** est disponible dans **<MyComponent />** :

```
import React from 'react';
import AuthContext from './AuthContext';
import MyComponent from './MyComponent';
function App() {
  // L'utilisateur est authentifié et nous obtenons le nom d'utilisateur
  const userName = 'john';
  return (
    <AuthContext.Provider value={userName}>
      <MyComponent />
    </AuthContext.Provider>
  );
};
export default App;
```

Maintenant, nous pouvons accéder à la valeur fournie dans n'importe quel composant de l'arborescence des composants en utilisant le hook **useContext()**, comme suit :

```
import React from 'react';
import AuthContext from './AuthContext';
function MyComponent() {
  const author = React.useContext(AuthContext);
  return (
    <div>
```

```

    Bienvenue {author}
  </div>
);
}
export default MyComponent;

```

Le composant affiche maintenant le texte *Bienvenue john*.

Gestion des listes avec React

Pour la gestion des listes, nous allons utiliser la méthode **map()** qui est utile lorsque vous devez manipuler une liste. La méthode **map()** crée un nouveau tableau contenant les résultats de l'appel d'une fonction sur chaque élément du tableau d'origine. Dans l'exemple suivant, chaque élément du tableau est multiplié par 2 :

```

const arr = [1, 2, 3, 4];
const resArr = arr.map(x => x * 2); // resArr = [2, 4, 6, 8]

```

La méthode **map()** dispose également d'un deuxième argument, l'index, qui est utile lors de la manipulation de listes en React. Les éléments de liste en React doivent avoir une clé unique qui est utilisée pour détecter les lignes qui ont été mises à jour, ajoutées ou supprimées.

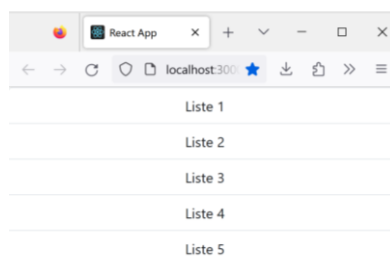
Le code d'exemple suivant présente un composant qui transforme un tableau d'entiers en un tableau d'éléments de liste et les affiche à l'intérieur de l'élément `ul` :

```

import React from 'react';
function MyList() {
  const data = [1, 2, 3, 4, 5];
  return (
    <div>
      <ul>
        {
          data.map((number, index) =>
            <li key={index}>Liste {number}</li>
          )
        }
      </ul>
    </div>
  );
}
export default MyList;

```

La capture d'écran suivante montre à quoi ressemble le composant lorsqu'il est rendu :

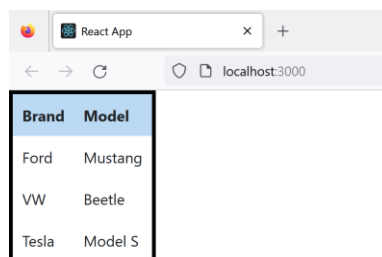


Si les données sont un tableau d'objets, il serait préférable de les présenter sous forme de tableau. Nous le faisons de manière similaire à la liste, mais cette fois nous mappions simplement le tableau

sur les lignes du tableau (éléments tr) et les rendons à l'intérieur de l'élément table, comme illustré dans le code de composant suivant :

```
import React from 'react';
function MyTable() {
  const data = [
    {brand: 'Ford', model: 'Mustang'},
    {brand: 'VW', model: 'Beetle'},
    {brand: 'Tesla', model: 'Model S'}
  ];
  return (
    <div>
      <table>
        <tbody>
          {
            data.map((item, index) =>
              <tr key={index}>
                <td>{item.brand}</td><td>{item.model}</td>
              </tr>
            )
          }
        </tbody>
      </table>
    </div>
  );
};
export default MyTable;
```

La capture d'écran suivante montre à quoi ressemble le composant lorsqu'il est rendu. Maintenant, vous devriez voir les données dans un tableau HTML :

A screenshot of a web browser window titled 'React App' at 'localhost:3000'. The browser displays a table with two columns, 'Brand' and 'Model'. The table contains three rows of data: Ford Mustang, VW Beetle, and Tesla Model S. The table is highlighted with a black border.

Brand	Model
Ford	Mustang
VW	Beetle
Tesla	Model S

Maintenant, nous avons appris comment gérer les données de liste en utilisant la méthode **map()** et comment les afficher, par exemple, dans un élément de tableau HTML.

Gestion des événements avec React

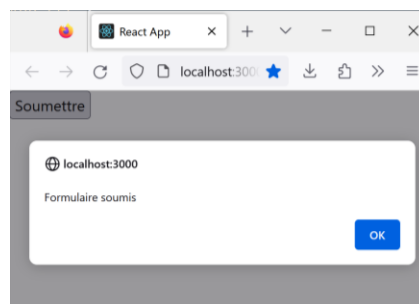
La gestion des événements dans React est similaire à la gestion des événements des éléments DOM. La différence par rapport à la gestion des événements HTML est que les noms d'événements utilisent la casse camelCase en React. Le code d'exemple suivant ajoute un écouteur d'événement à un bouton et affiche une alerte lorsque le bouton est pressé :

```
import React from 'react';
function MyButton() {
  // Cela est appelé lorsque le bouton est pressé
  const buttonPressed = () => {
    alert('Bouton pressé');
  }
  return (
    <div>
      <button onClick={buttonPressed}>Appuie-moi</button>
    </div>
  );
};
export default MyButton;
```

En React, vous ne pouvez pas faire « return false » à partir du gestionnaire d'événements pour empêcher le comportement par défaut (envoyer les données dans une nouvelle page). À la place, vous devez appeler la méthode **preventDefault()**. Dans l'exemple suivant, nous utilisons un élément de formulaire et nous voulons empêcher l'envoi du formulaire :

```
import React from 'react';
function MyForm() {
  // Cela est appelé lorsque le formulaire est soumis
  const handleSubmit = (event) => {
    event.preventDefault(); // Empêche le comportement par défaut
    alert('Formulaire soumis');
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="submit" value="Soumettre" />
    </form>
  );
};
export default MyForm;
```

Maintenant, lorsque vous appuyez sur le bouton Soumettre, vous pouvez voir l'alerte et le formulaire ne sera pas soumis.



Gestion des formulaires avec React

La manipulation de formulaires est un peu différente avec React. Un formulaire HTML se dirigera vers la page suivante lorsqu'il est soumis. Souvent, nous voulons invoquer une fonction JavaScript qui a

accès aux données du formulaire après la soumission et éviter de naviguer vers la page suivante. Nous avons déjà expliqué comment éviter la soumission en utilisant **preventDefault()**.

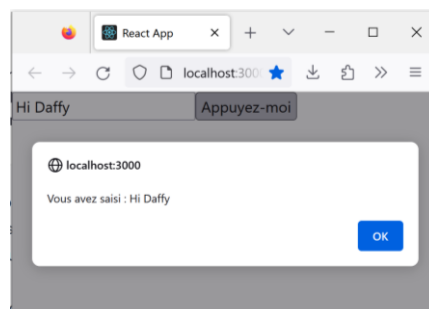
La manipulation des formulaires avec React

Commençons d'abord par créer un formulaire minimaliste avec un champ de saisie et un bouton de soumission. Pour obtenir la valeur du champ de saisie, nous utilisons le gestionnaire d'événements **onChange**. Nous utilisons le hook **useState** pour créer une variable d'état appelée « texte ». Lorsque la valeur du champ de saisie est modifiée, la nouvelle valeur est enregistrée dans l'état. Cela s'appelle également un composant contrôlé car les données du formulaire sont gérées par React.

L'instruction "**setText(event.target.value)**" récupère la valeur du champ de saisie et l'enregistre dans l'état. Enfin, nous afficherons la valeur saisie lorsque l'utilisateur appuie sur le bouton Soumettre. Voici le code source de notre premier formulaire :

```
import React, { useState } from 'react';
function MyForm() {
  const [texte, setText] = useState("");
  // Enregistrer la valeur de l'élément de saisie dans l'état lorsqu'elle a été modifiée
  const champModifie = (event) => {
    setText(event.target.value);
  }
  const handleSubmit = (event) => {
    alert(`Vous avez saisi : ${texte}`);
    event.preventDefault();
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" onChange={champModifie} value={texte} />
      <input type="submit" value="Appuyez-moi" />
    </form>
  );
}
export default MyForm;
```

Voici une capture d'écran de notre composant de formulaire après l'appui sur le bouton Soumettre :



Vous pouvez également écrire une fonction de gestionnaire d'événements **onChange** en ligne en utilisant JSX, comme indiqué dans l'exemple suivant :

```
return (
  <form onSubmit={handleSubmit}>
    <input
```

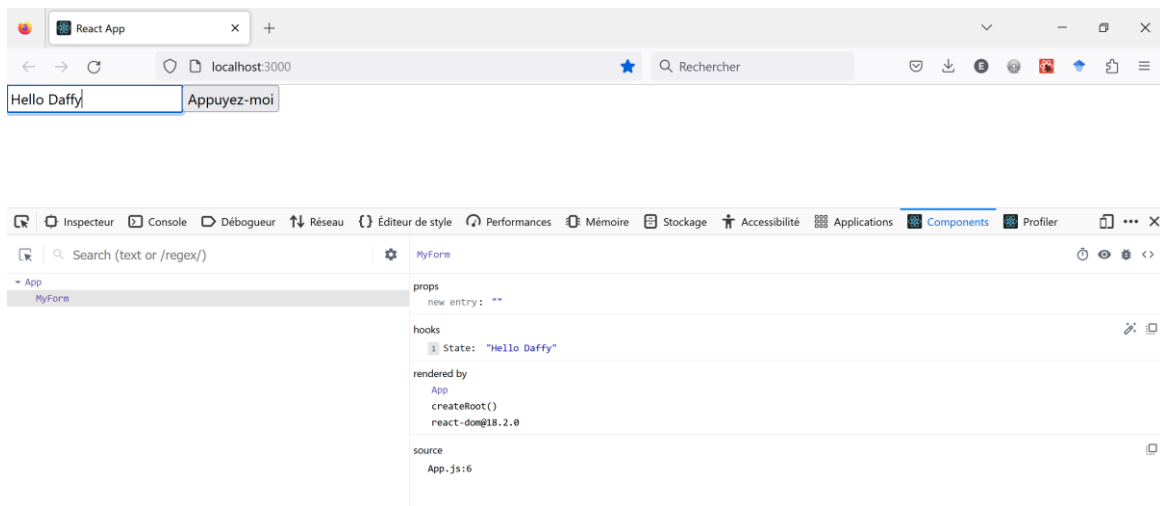
```

    type="text"
    onChange={event => setTexte(event.target.value)}
    value={texte}
  />
  <input type="submit" value="Appuyez-moi" />
</form>
);

```

C'est maintenant le bon moment pour découvrir les outils de développement React, qui sont utiles pour déboguer les applications React. Si nous ouvrons les outils de développement React avec notre application de formulaire React et que nous saisissons quelque chose dans le champ de saisie, nous pouvons voir comment la valeur de l'état change et nous pouvons inspecter la valeur actuelle des props et de l'état¹.

La capture d'écran suivante montre comment l'état change lorsque nous saisissons quelque chose dans le champ de saisie :



En général, nous avons plus d'un champ de saisie dans le formulaire. Voyons comment gérer cela en utilisant un état objet. Tout d'abord, nous introduisons un état appelé « **utilisateur** » en utilisant le crochet **useState**, comme le montre l'extrait de code suivant. L'état "utilisateur" est un objet avec trois attributs : "prenom", "nom" et "email" :

```

const [utilisateur, setUtilisateur] = useState({
  prenom: "",
  nom: "",
  email: ""
});

```

Une façon de gérer plusieurs champs de saisie consiste à ajouter autant de gestionnaires de changement que nous avons de champs de saisie, mais cela crée beaucoup de code redondant que nous voulons éviter. Par conséquent, nous ajoutons des attributs "**name**" à nos champs de saisie. Nous pouvons les utiliser dans le gestionnaire de changement pour identifier quel champ de saisie déclenche le gestionnaire de changement. La valeur de l'attribut "**name**" de l'élément de saisie doit être identique au nom de la propriété de l'objet d'état dans laquelle nous voulons enregistrer la valeur,

¹ Faut installer l'extension React devTools dans votre navigateur

et la valeur de l'attribut **"value"** doit être **"objet.propriété"**, par exemple, dans le premier élément de saisie du prénom. Le code est illustré ici :

```
<input type="text" name="nom" onChange={champModifie} value={utilisateur.nom} />
```

Le gestionnaire de changement de saisie ressemble maintenant à ceci. Si le champ de saisie qui déclenche le gestionnaire est le champ du prénom, alors **"event.target.name"** est **"prenom"**, et la valeur saisie sera enregistrée dans le champ **"prenom"** de l'objet d'état. Ici, nous utilisons également la notation de propagation d'objet ("object spread notation") qui a été introduite dans le lab précédent. De cette manière, nous pouvons gérer tous les champs de saisie avec un seul gestionnaire de changement :

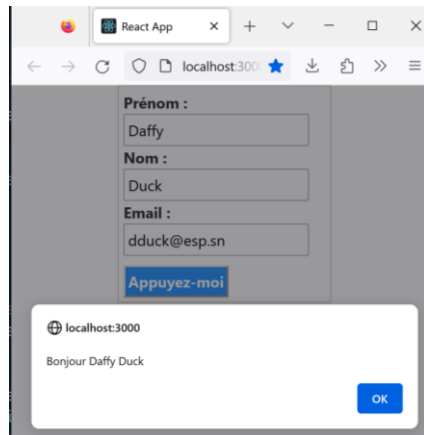
```
const champModifie = (event) => {  
  setUtilisateur({ ...utilisateur, [event.target.name]: event.target.value });  
}
```

Voici le code source complet du composant :

```
import React, { useState } from 'react';  
function MyForm() {  
  const [utilisateur, setUtilisateur] = useState({  
    prenom: "",  
    nom: "",  
    email: ""  
  });  
  
  // Enregistrer la valeur de la zone de saisie dans l'état lorsqu'elle est modifiée  
  const champModifie = (event) => {  
    setUtilisateur({ ...utilisateur, [event.target.name]: event.target.value });  
  }  
  
  const handleSubmit = (event) => {  
    alert(`Bonjour ${utilisateur.prenom} ${utilisateur.nom}`);  
    event.preventDefault();  
  }  
  
  return (  
    <form onSubmit={handleSubmit}>  
      <label>Prénom :</label>  
      <input type="text" name="prenom" onChange={champModifie} value={utilisateur.prenom}  
    /><br />  
      <label>Nom :</label>  
      <input type="text" name="nom" onChange={champModifie} value={utilisateur.nom} /><br />  
      <label>Email :</label>  
      <input type="email" name="email" onChange={champModifie} value={utilisateur.email} /><br />  
      <input type="submit" value="Appuyez-moi" />  
    </form>  
  );  
}
```

```
export default MyForm;
```

Voici une capture d'écran de notre composant de formulaire après l'appui sur le bouton Soumettre :



L'exemple précédent peut également être implémenté en utilisant des états séparés au lieu d'un seul état et d'un objet. L'extrait de code suivant le démontre. Maintenant, nous avons trois états, et dans le gestionnaire d'événements onChange de l'élément de saisie, nous appelons la fonction de mise à jour appropriée pour enregistrer les valeurs dans les états. Dans ce cas, nous n'avons pas besoin de l'attribut "name" de l'élément de saisie :

```
import React, { useState } from 'react';
function MyForm() {
  const [prenom, setPrenom] = useState("");
  const [nom, setNom] = useState("");
  const [email, setEmail] = useState("");
  const handleSubmit = (event) => {
    alert(`Bonjour ${prenom} ${nom}`);
    event.preventDefault();
  }
  return (
    <form onSubmit={handleSubmit}>
      <label>Prénom :</label>
      <input onChange={e => setPrenom(e.target.value)} value={prenom} /><br />
      <label>Nom :</label>
      <input onChange={e => setNom(e.target.value)} value={nom} /><br />
      <label>Email :</label>
      <input onChange={e => setEmail(e.target.value)} value={email} /><br />
      <input type="submit" value="Appuyez-moi" />
    </form>
  );
}
```

export default MyForm;

Maintenant, nous savons comment gérer les formulaires avec React, et nous utiliserons ces compétences plus tard lorsque nous mettrons en œuvre notre tiers présentation.