**Introduction**

Dans ce didacticiel, vous partagerez l'état entre plusieurs composants à l'aide de [React context](https://reactjs.org/docs/context.html) . React context est une interface pour partager des informations avec d'autres [composants](https://www.digitalocean.com/community/tutorials/how-to-create-custom-components-in-react) sans transmettre explicitement les données en tant que [props](https://www.digitalocean.com/community/tutorials/how-to-customize-react-components-with-props) . Cela signifie que vous pouvez partager des informations entre un composant parent et un composant enfant profondément imbriqué, ou stocker des données à l'échelle du site en un seul endroit et y accéder n'importe où dans l'application. Vous pouvez même mettre à jour les données des composants imbriqués en fournissant des fonctions de mise à jour avec les données.

Le contexte React est suffisamment flexible pour être utilisé comme système de gestion d'état centralisé pour votre projet, ou vous pouvez l'étendre à de plus petites sections de votre application. Avec le contexte, vous pouvez partager des données dans l'application sans aucun outil tiers supplémentaire et avec une petite quantité de configuration. Cela fournit une alternative plus légère aux outils comme [Redux](https://redux.js.org/) , qui peuvent aider avec des applications plus importantes mais peuvent nécessiter trop de configuration pour les projets de taille moyenne.

Tout au long de ce didacticiel, vous utiliserez le contexte pour créer une application qui utilise des ensembles de données communs à différents composants. Pour illustrer cela, vous allez créer un site Web où les utilisateurs peuvent créer des salades personnalisées. Le site Web utilisera le contexte pour stocker les informations sur les clients, les articles préférés et les salades personnalisées. Vous accéderez ensuite à ces données et les mettrez à jour dans toute l'application sans transmettre les données via des props. À la fin de ce didacticiel, vous apprendrez à utiliser le contexte pour stocker des données à différents niveaux du projet et à accéder aux données et à les mettre à jour dans des composants imbriqués.

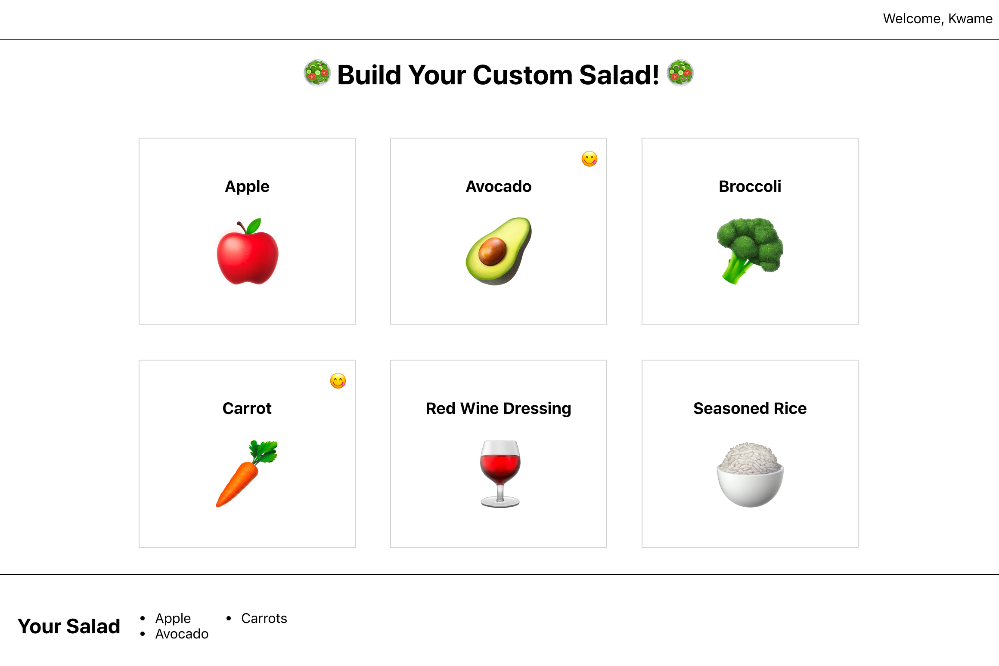
**Conditions préalables**

* Vous aurez besoin d'un environnement de développement exécutant [Node.js](https://nodejs.org/en/about/) ; ce tutoriel a été testé sur Node.js version 10.20.1 et npm version 6.14.4. Pour l'installer sur macOS ou Ubuntu 18.04, suivez les étapes de [Comment installer Node.js et créer un environnement de développement local sur macOS](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-and-create-a-local-development-environment-on-macos) ou la section **Installation à l'aide d'un PPA** de [Comment installer Node.js sur Ubuntu 18.04](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-18-04) .
* Un environnement de développement React configuré avec [Create React App](https://github.com/facebook/create-react-app) , avec le passe-partout non essentiel supprimé. Pour le configurer, suivez l' [**Étape 1 — Création d'un projet vide** du didacticiel Comment gérer l'état sur les composants de classe React](https://www.digitalocean.com/community/tutorials/how-to-manage-state-on-react-class-components#step-1-%E2%80%94-creating-an-empty-project) . Ce didacticiel utilisera state-context-tutorialcomme nom de projet.
* Vous aurez également besoin d'une connaissance de base de JavaScript, que vous pouvez trouver dans [How To Code in JavaScript](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-javascript) , ainsi que d'une connaissance de base de HTML et CSS. Une ressource utile pour HTML et CSS est le [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML) .
* Vous utiliserez les composants React, le useStateHook et le useReducerHook, que vous pouvez découvrir dans nos tutoriels [How To Create Custom Components in React](https://www.digitalocean.com/community/tutorials/how-to-create-custom-components-in-react) et [How To Manage State with Hooks on React Components](https://www.digitalocean.com/community/tutorials/how-to-manage-state-with-hooks-on-react-components) .

**Étape 1 - Construire la base de votre application**

Dans cette étape, vous allez construire la structure générale de votre préparateur de salade personnalisé. Vous allez créer des composants pour afficher les garnitures possibles, une liste des garnitures sélectionnées et des informations sur le client. Au fur et à mesure que vous créez l'application avec des données statiques, vous découvrirez comment différentes informations sont utilisées dans divers composants et comment identifier les éléments de données qui seraient utiles dans un contexte.

Voici un exemple de l'application que vous allez créer :



Notez qu'il existe des informations que vous pourriez avoir besoin d'utiliser dans les composants. Par exemple, le nom d'utilisateur (qui pour cet exemple est **Kwame** ) affiche les données utilisateur dans une zone de navigation, mais vous pouvez également avoir besoin d'informations utilisateur pour identifier les éléments favoris ou pour une page de paiement. Les informations utilisateur devront être accessibles par n'importe quel composant de l'application. En regardant le constructeur de salade lui-même, chaque ingrédient de salade devra pouvoir mettre à jour la liste **Votre salade** en bas de l'écran, vous devrez donc stocker et mettre à jour ces données à partir d'un emplacement accessible à chaque composant. .

Commencez par coder en dur toutes les données afin de pouvoir élaborer la structure de votre application. Plus tard, vous ajouterez le contexte en commençant à l'étape suivante. Le contexte fournit le plus de valeur au fur et à mesure que les applications commencent à se développer, donc dans cette étape, vous allez créer plusieurs composants pour montrer comment le contexte fonctionne dans une arborescence de composants. Pour les composants ou les bibliothèques plus petits, vous pouvez souvent utiliser des composants d'encapsulation et des techniques de gestion d'état de niveau inférieur, telles que [React Hooks](https://www.digitalocean.com/community/tutorials/how-to-manage-state-with-hooks-on-react-components) et [la gestion basée sur les classes](https://www.digitalocean.com/community/tutorials/how-to-manage-state-on-react-class-components) .

Étant donné que vous créez une petite application avec plusieurs composants, installez [JSS](https://cssinjs.org/) pour vous assurer qu'il n'y aura pas de conflits de noms de classe et pour pouvoir ajouter des styles dans le même fichier qu'un composant. Pour en savoir plus sur JSS, consultez [Styling React Components](https://www.digitalocean.com/community/tutorials/how-to-style-react-components) .

Exécutez la commande suivante :

1. npm install react-jss

Copie

npm installera le composant, et une fois terminé, vous verrez un message comme celui-ci :

Output

+ react-jss@10.3.0

added 27 packages from 10 contributors, removed 10 packages andaudited 1973 packages in 15.507s

Maintenant que JSS est installé, considérez les différents composants dont vous aurez besoin. En haut de la page, vous aurez un composant Navigation pour stocker le message de bienvenue. Le composant suivant sera le SaladMaker lui-même. Cela contiendra le titre avec le constructeur et la liste **Your Salad** en bas. La section avec les ingrédients sera un composant séparé appelé le SaladBuilder, imbriqué à l'intérieur SaladMaker. Chaque ingrédient sera une instance d'un composant SaladItem. Enfin, la liste du bas sera un composant appelé SaladSummary.

**Remarque :** Les composants n'ont pas besoin d'être divisés de cette façon. Au fur et à mesure que vous travaillez sur vos applications, votre structure changera et évoluera à mesure que vous ajouterez plus de fonctionnalités. Cet exemple est destiné à vous donner une structure pour explorer comment le contexte affecte les différents composants de l'arborescence.

Maintenant que vous avez une idée des composants dont vous aurez besoin, créez un répertoire pour chacun :

1. mkdir src/components/Navigation
2. mkdir src/components/SaladMaker
3. mkdir src/components/SaladItem
4. mkdir src/components/SaladBuilder
5. mkdir src/components/SaladSummary

Copie

Ensuite, construisez les composants de haut en bas en commençant par Navigation. Commencez par ouvrir le fichier du composant dans un éditeur de texte :

1. nano src/components/Navigation/Navigation.js

Copie

Créez un composant appelé Navigationet ajoutez un style pour lui donner Navigationune bordure et un rembourrage :

état-contexte-tutoriel/src/components/Navigation/Navigation.js

import React from 'react';

import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({

wrapper: {

borderBottom: 'black solid 1px',

padding: [15, 10],

textAlign: 'right',

}

});

export default function Navigation() {

const classes = useStyles();

return(

<div className={classes.wrapper}>

Welcome, Kwame

</div>

)

}

Copie

Puisque vous utilisez JSS, vous pouvez créer des objets de style directement dans le composant plutôt que dans un fichier CSS. Le wrapper div aura un rembourrage, une bordure solid black et alignera le texte à droite avec textAlign.

Enregistrez et fermez le fichier. Ensuite, ouvrez App.js, qui est la racine du projet :

1. nano src/components/App/App.js

Copie

Importez le Navigationcomposant et affichez-le dans des balises vides en ajoutant les lignes en surbrillance :

état-contexte-tutoriel/src/components/App/App.js

import React from 'react';

import Navigation from '../Navigation/Navigation';

function App() {

return (

<>

<Navigation />

</>

);

}

export default App;

Copie

Enregistrez et fermez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et vous verrez la barre de navigation :



Considérez la barre de navigation comme un *composant global*, car dans cet exemple, elle sert de composant modèle qui sera réutilisé sur chaque page.

Le composant suivant sera SaladMaker. Il s'agit d'un composant qui ne s'affichera que sur certaines pages ou dans certains états.

Ouvrez SaladMaker.js dans votre éditeur de texte :

1. nano src/components/SaladMaker/SaladMaker.js

Copie

Créez un composant qui a une balise <h1> avec l'en-tête :

état-contexte-tutoriel/src/components/SaladMaker/SaladMaker.js

import React from 'react';

import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({

wrapper: {

textAlign: 'center',

}

});

export default function SaladMaker() {

const classes = useStyles();

return(

<>

<h1 className={classes.wrapper}>

<span role="img" aria-label="salad">🥗 </span>

Build Your Custom Salad!

<span role="img" aria-label="salad"> 🥗</span>

</h1>

</>

)

}

Copie

Dans ce code, vous utilisez textAlign pour centrer le composant sur la page. Les attributs role et aria-label de l' élément span faciliteront l'accessibilité à l'aide [d'Applications Internet Riches Accessibles (ARIA)](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) .

Enregistrez et fermez le fichier. Ouvrez App.js pour afficher le composant :

1. nano src/components/App/App.js

Copie

Importez SaladMaker et affichez après le composant  Navigation:

état-contexte-tutoriel/src/components/App/App.js

import React from 'react';

import Navigation from '../Navigation/Navigation';

import SaladMaker from '../SaladMaker/SaladMaker';

function App() {

return (

<>

<Navigation />

<SaladMaker />

</>

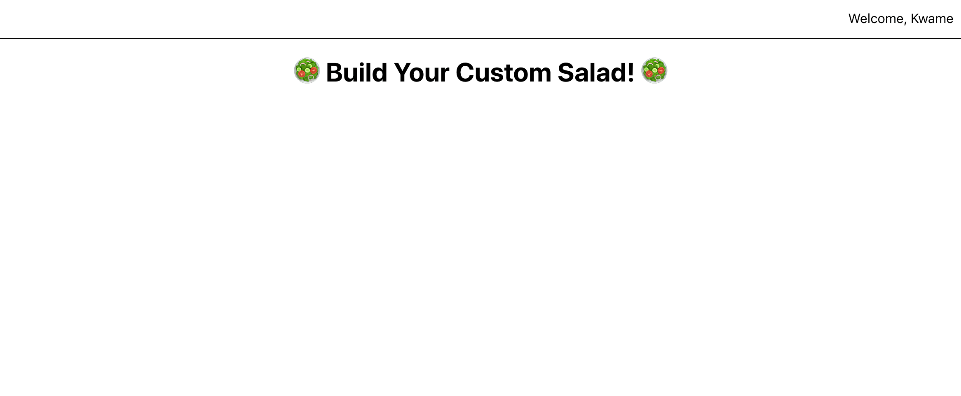
);

}

export default App;

Copie

Enregistrez et fermez le fichier. Lorsque vous le ferez, la page se rechargera et vous verrez l'en-tête :



Ensuite, créez un composant appelé SaladItem. Ce sera une carte pour chaque ingrédient individuel.

Ouvrez le fichier dans votre éditeur de texte :

1. nano src/components/SaladItem/SaladItem.js

Copie

Ce composant comportera trois parties : le nom de l'élément, une icône indiquant si l'élément est un favori de l'utilisateur et un emoji placé à l'intérieur d'un bouton qui ajoutera l'élément à la salade en un clic. Ajoutez les lignes suivantes à SaladItem.js:

état-contexte-tutoriel/src/components/SaladItem/SaladItem.js

import React from 'react';

import PropTypes from 'prop-types';

import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({

add: {

background: 'none',

border: 'none',

cursor: 'pointer',

},

favorite: {

fontSize: 20,

position: 'absolute',

top: 10,

right: 10,

},

image: {

fontSize: 80

},

wrapper: {

border: 'lightgrey solid 1px',

margin: 20,

padding: 25,

position: 'relative',

textAlign: 'center',

textTransform: 'capitalize',

width: 200,

}

});

export default function SaladItem({ image, name }) {

const classes = useStyles();

const favorite = true;

return(

<div className={classes.wrapper}>

<h3>

{name}

</h3>

<span className={classes.favorite} aria-label={favorite ? 'Favorite' : 'Not Favorite'}>

{favorite ? '😋' : ''}

</span>

<button className={classes.add}>

<span className={classes.image} role="img" aria-label={name}>{image}</span>

</button>

</div>

)

}

SaladItem.propTypes = {

image: PropTypes.string.isRequired,

name: PropTypes.string.isRequired,

}

Copie

Les image et name seront des props. Le code utilise l’opérateur [ternaires](https://www.digitalocean.com/community/tutorials/how-to-write-conditional-statements-in-javascript#ternary-operator) pour déterminer la valeur de la variable favorite conditionnellement si l'icône favorite apparaît ou non. La variable favorite sera ensuite déterminée avec le contexte dans le cadre du profil de l'utilisateur. Pour l'instant, réglez-le sur true. Le style placera l'icône de favori dans le coin supérieur droit de la carte et supprimera la bordure et l'arrière-plan par défaut du bouton. La classe wrapper ajoutera une petite bordure et transformera une partie du texte. Enfin, [PropTypes](https://www.digitalocean.com/community/tutorials/how-to-customize-react-components-with-props#step-3-%E2%80%94-creating-predictable-props-with-proptypes-and-defaultprops) ajoute un système de typage faible pour fournir une certaine application pour s'assurer que le mauvais type d'accessoire n'est pas passé.

Enregistrez et fermez le fichier. Maintenant, vous devrez rendre les différents éléments. Vous ferez cela avec un composant appelé SaladBuilder, qui contiendra une liste d'éléments qu'il convertira en une série de composants  SaladItem :

Ouvrez SaladBuilder :

1. nano src/components/SaladBuilder/SaladBuilder.js

Copie

S'il s'agissait d'une application de production, ces données proviendraient souvent d'une interface de programmation d'application (API). Mais pour l'instant, utilisez une liste d'ingrédients codée en dur :

state-context-tutorial/src/components/SaladBuilder/SaladBuilder.js

import React from 'react';

import SaladItem from '../SaladItem/SaladItem';

import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({

wrapper: {

display: 'flex',

flexWrap: 'wrap',

padding: [10, 50],

justifyContent: 'center',

}

});

const ingredients = [

{

image: '🍎',

name: 'apple',

},

{

image: '🥑',

name: 'avocado',

},

{

image: '🥦',

name: 'broccoli',

},

{

image: '🥕',

name: 'carrot',

},

{

image: '🍷',

name: 'red wine dressing',

},

{

image: '🍚',

name: 'seasoned rice',

},

];

export default function SaladBuilder() {

const classes = useStyles();

return(

<div className={classes.wrapper}>

{

ingredients.map(ingredient => (

<SaladItem

key={ingredient.name}

image={ingredient.image}

name={ingredient.name}

/>

))

}

</div>

)

}

Copie

Cet extrait utilise la méthode array [map()](https://www.digitalocean.com/community/tutorials/how-to-use-array-methods-in-javascript-iteration-methods#map()) pour mapper chaque élément de la liste, en transmettant les props name et image comme props à un composant SaladItem. Assurez-vous d' [ajouter un key à chaque élément lorsque vous mappez](https://www.digitalocean.com/community/tutorials/how-to-create-react-elements-with-jsx#step-4-%E2%80%94-mapping-over-data-to-create-elements) . Le style de ce composant ajoute un affichage flex pour la mise en [page flexbox](https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox) , enveloppe les composants et les centre.

Enregistrez et fermez le fichier.

Enfin, rendez le composant SaladMaker afin qu'il apparaisse dans la page.

Ouvrez SaladMaker:

1. nano src/components/SaladMaker/SaladMaker.js

Copie

Ensuite, importez SaladBuilder et affichez après l'en-tête :

état-contexte-tutoriel/src/components/SaladMaker/SaladMaker.js

import React from 'react';

import { createUseStyles } from 'react-jss';

import SaladBuilder from '../SaladBuilder/SaladBuilder';

const useStyles = createUseStyles({

wrapper: {

textAlign: 'center',

}

});

export default function SaladMaker() {

const classes = useStyles();

return(

<>

<h1 className={classes.wrapper}>

<span role="img" aria-label="salad">🥗 </span>

Build Your Custom Salad!

<span role="img" aria-label="salad"> 🥗</span>

</h1>

<SaladBuilder />

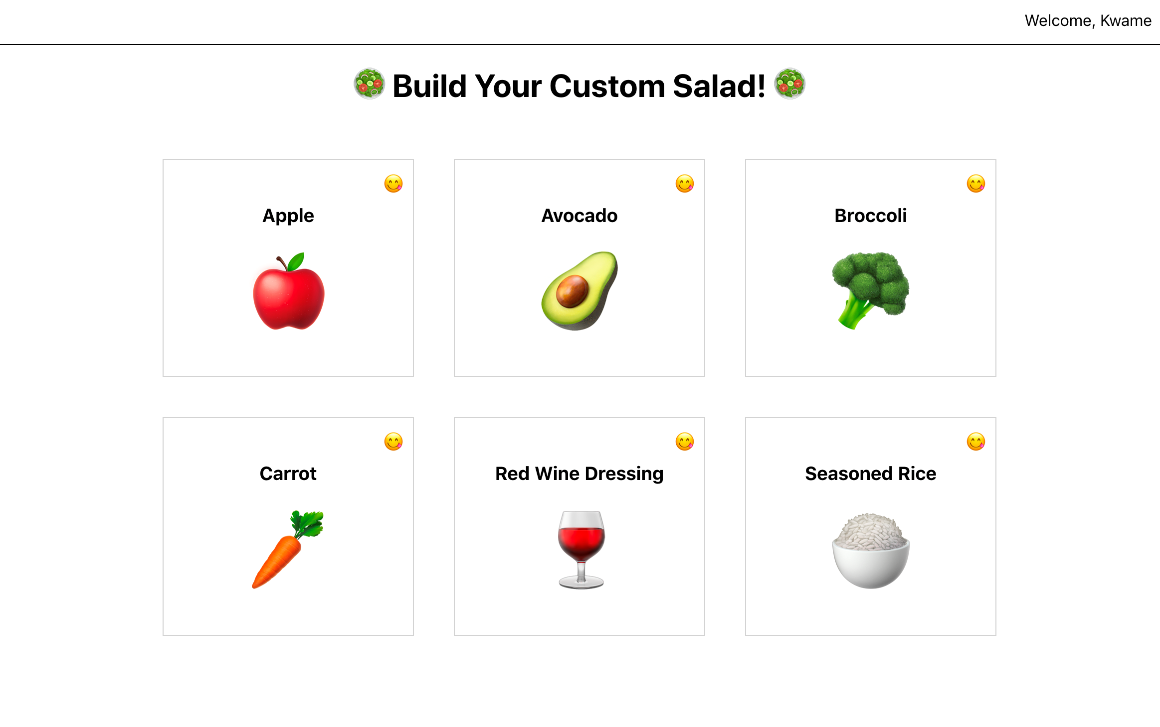
</>

)

}

Copie

Enregistrez et fermez le fichier. Lorsque vous le ferez, la page se rechargera et vous trouverez le contenu :



La dernière étape consiste à ajouter le récapitulatif de la salade en cours. Ce composant affichera une liste d'éléments qu'un utilisateur a sélectionnés. Pour l'instant, vous allez coder en dur les éléments. Vous les mettrez à jour avec le contexte à l'étape 3.

Ouvrez SaladSummary dans votre éditeur de texte :

1. nano src/components/SaladSummary/SaladSummary.js

Copie

Le composant sera un en-tête et une liste non triée d'éléments. Vous utiliserez flexbox pour les faire envelopper :

état-contexte-tutoriel/src/components/SaladSummary/SaladSummary.jss

import React from 'react';

import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({

list: {

display: 'flex',

flexDirection: 'column',

flexWrap: 'wrap',

maxHeight: 50,

'& li': {

width: 100

}

},

wrapper: {

borderTop: 'black solid 1px',

display: 'flex',

padding: 25,

}

});

export default function SaladSummary() {

const classes = useStyles();

return(

<div className={classes.wrapper}>

<h2>Your Salad</h2>

<ul className={classes.list}>

<li>Apple</li>

<li>Avocado</li>

<li>Carrots</li>

</ul>

</div>

)

}

Copie

Enregistrez le fichier. Ouvrez ensuite SaladMaker pour afficher l'élément :

1. nano src/components/SaladMaker/SaladMaker.js

Copie

Importer et ajouter SaladSummary après le SaladBuilder :

état-contexte-tutoriel/src/components/SaladMaker/SaladMaker.js

import React from 'react';

import { createUseStyles } from 'react-jss';

import SaladBuilder from '../SaladBuilder/SaladBuilder';

import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({

wrapper: {

textAlign: 'center',

}

});

export default function SaladMaker() {

const classes = useStyles();

return(

<>

<h1 className={classes.wrapper}>

<span role="img" aria-label="salad">🥗 </span>

Build Your Custom Salad!

<span role="img" aria-label="salad"> 🥗</span>

</h1>

<SaladBuilder />

<SaladSummary />

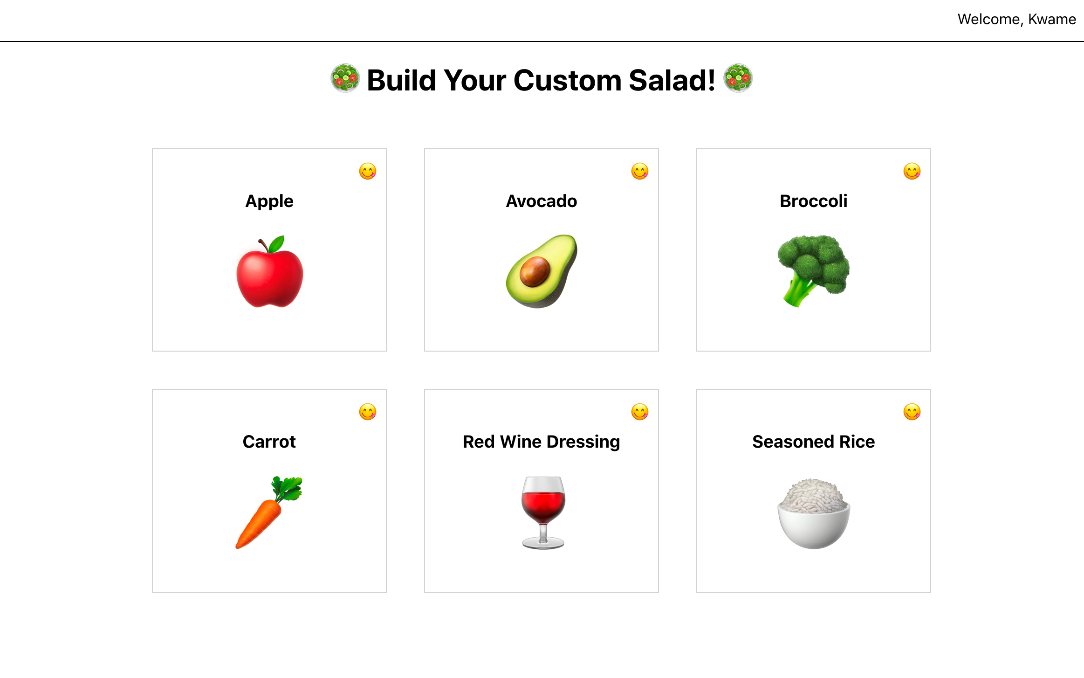
</>

)

}

Copie

Enregistrez et fermez le fichier. Lorsque vous le ferez, la page se rafraîchira et vous trouverez l'application complète :



Il y a des données partagées dans toute l'application. Le composant Navigation et le composant SaladItem ont tous deux besoin de connaître quelque chose sur l'utilisateur : son nom et sa liste de favoris. Le SaladItem doit également mettre à jour les données accessibles dans le composant SaladSummary. Les composants partagent des ancêtres communs, mais la transmission des données à travers l'arborescence serait difficile et sujette aux erreurs.

C'est là qu'intervient le contexte. Vous pouvez déclarer les données dans un parent commun, puis y accéder ultérieurement sans les transmettre explicitement dans la hiérarchie des composants.

Dans cette étape, vous avez créé une application pour permettre à l'utilisateur de créer une salade à partir d'une liste d'options. Vous avez créé un ensemble de composants qui doivent accéder ou mettre à jour des données contrôlées par d'autres composants. Dans l'étape suivante, vous utiliserez le contexte pour stocker des données et y accéder dans des composants enfants.

**Étape 2 - Fournir des données à partir d'un composant racine**

Dans cette étape, vous utiliserez le contexte pour stocker les informations client à la racine du composant. Vous allez créer un contexte personnalisé, puis utiliser un composant d'habillage spécial appelé [Provider](https://reactjs.org/docs/context.html#contextprovider) qui stockera les informations à la racine du projet. Vous utiliserez ensuite le hook [useContext](https://reactjs.org/docs/hooks-reference.html#usecontext) pour vous connecter au fournisseur dans les composants imbriqués afin de pouvoir afficher les informations statiques. À la fin de cette étape, vous serez en mesure de fournir des magasins d'informations centralisés et d'utiliser les informations stockées dans un contexte dans de nombreux composants différents.

Le contexte dans sa forme la plus élémentaire est une interface de partage d'informations. De nombreuses applications ont des informations universelles qu'elles doivent partager dans l'ensemble de l'application, telles que les préférences de l'utilisateur, les informations sur les thèmes et les modifications d'application à l'échelle du site. Avec le contexte, vous pouvez stocker ces informations au niveau racine, puis y accéder n'importe où. Puisque vous définissez les informations dans un parent, vous savez qu'elles seront toujours disponibles et qu'elles seront toujours à jour.

Pour ajouter un contexte, créez un nouveau répertoire appelé User :

1. mkdir src/components/User

Copie

User ne va pas être un composant traditionnel, en ce sens que vous allez l'utiliser à la fois comme composant et comme élément de données pour un hook spécial appelé useContext. Pour l'instant, conservez la structure de fichiers plats, mais si vous utilisez beaucoup de contextes, cela peut valoir la peine de les déplacer vers une autre structure de répertoires.

Ensuite, ouvrez User.js dans votre éditeur de texte :

1. nano src/components/User/User.js

Copie

Dans le fichier, importez la fonction createContext depuis React, puis exécutez la fonction et exportez le résultat :

state-context-tutorial/src/components/User/User.js

import { createContext } from 'react';

const UserContext = createContext();

export default UserContext;

Copie

En exécutant la fonction, vous avez enregistré le contexte. Le résultat, UserContext, est ce que vous utiliserez dans vos composants.

Enregistrez et fermez le fichier.

L'étape suivante consiste à appliquer le contexte à un ensemble d'éléments. Pour ce faire, vous utiliserez un composant appelé Provider. Le Provider est un composant qui définit les données, puis encapsule certains composants enfants. Tous les composants enfants encapsulés auront accès aux données du Provider avec le hook useContext.

Étant donné que les données utilisateur seront constantes tout au long du projet, placez-les aussi haut que possible dans l'arborescence des composants. Dans cette application, vous le mettrez au niveau racine dans le composant App :

Ouvrez App :

1. nano src/components/App/App.js

Copie

Ajoutez les lignes de code en surbrillance suivantes pour importer le contexte et transmettre les données :

état-contexte-tutoriel/src/components/App/App.js

import React from 'react';

import Navigation from '../Navigation/Navigation';

import SaladMaker from '../SaladMaker/SaladMaker';

import UserContext from '../User/User';

const user = {

name: 'Kwame',

favorites: [

'avocado',

'carrot'

]

}

function App() {

return (

<UserContext.Provider value={user}>

<Navigation />

<SaladMaker />

</UserContext.Provider>

);

}

export default App;

Copie

Dans une application typique, vous récupérez les données utilisateur ou les stockez lors d'un rendu côté serveur. Dans ce cas, vous avez codé en dur certaines données que vous pourriez recevoir d'une API. Vous avez créé un [objet](https://www.digitalocean.com/community/tutorials/understanding-objects-in-javascript) appelé user qui contient le nom d'utilisateur sous forme de chaîne et un [tableau](https://www.digitalocean.com/community/tutorials/understanding-arrays-in-javascript) d'ingrédients favoris.

Ensuite, vous avez importé le UserContext, puis enveloppé Navigation et SaladMaker avec un composant appelé le UserContext.Provider. Remarquez comment, dans ce cas UserContext, agit comme un composant React standard. Ce composant prendra une seule prop appelé value. Cette prop sera les données que vous souhaitez partager, qui dans ce cas est l'objet user.

Enregistrez et fermez le fichier. Désormais, les données sont disponibles dans toute l'application. Cependant, pour utiliser les données, vous devrez à nouveau importer et accéder au contexte.

Maintenant que vous avez défini le contexte, vous pouvez commencer à remplacer les données codées en dur dans votre composant par des valeurs dynamiques. Commencez par remplacer le nom codé en dur dans Navigation par les données utilisateur que vous avez définies avec UserContext.Provider.

Ouvrez Navigation.js:

1. nano src/components/Navigation/Navigation.js

Copie

À l' intérieur de Navigation, importez le Hook useContext depuis React et UserContext depuis le répertoire du composant. Appelez ensuite useContext utilisant UserContext comme argument. Contrairement au UserContext.Provider, vous n'avez pas besoin de rendre UserContext dans le [JSX](https://www.digitalocean.com/community/tutorials/how-to-create-react-elements-with-jsx) . Le hook renverra les données que vous avez fournies dans la prop value. Enregistrez les données dans une nouvelle variable appelée user, qui est un objet contenant name et favorites. Vous pouvez alors remplacer le nom codé en dur par user.name:

état-contexte-tutoriel/src/components/Navigation/Navigation.js

import React, { useContext } from 'react';

import { createUseStyles } from 'react-jss';

import UserContext from '../User/User';

const useStyles = createUseStyles({

wrapper: {

outline: 'black solid 1px',

padding: [15, 10],

textAlign: 'right',

}

});

export default function Navigation() {

const user = useContext(UserContext);

const classes = useStyles();

return(

<div className={classes.wrapper}>

Welcome, {user.name}

</div>

)

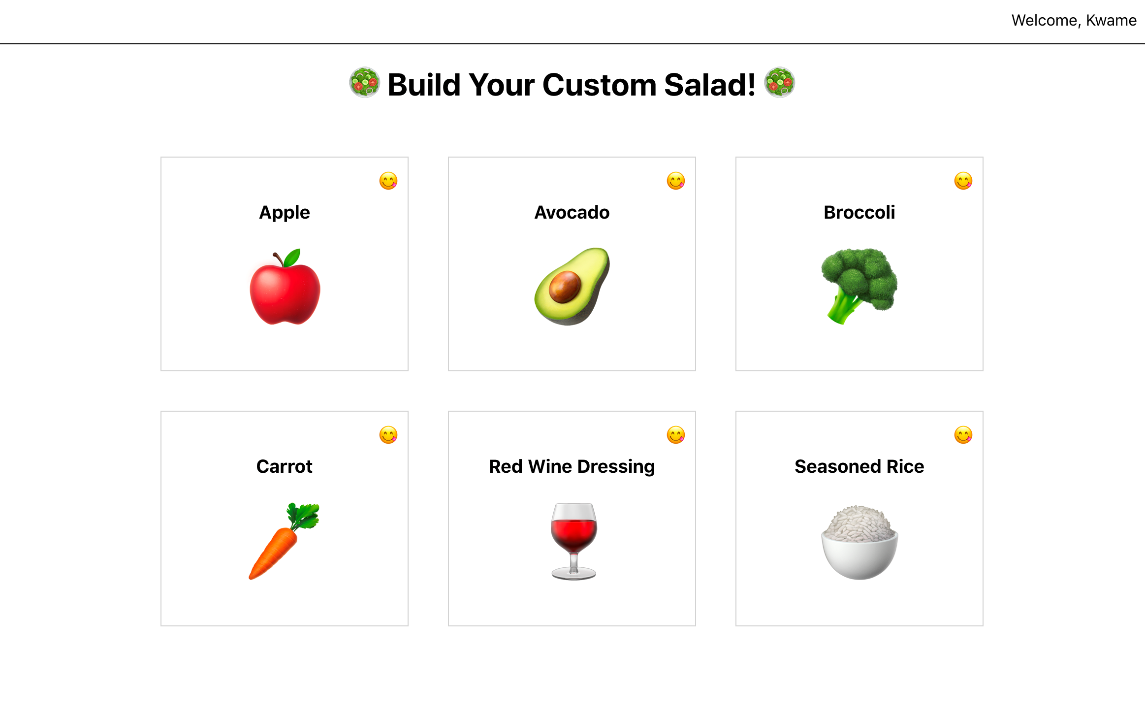
}

Copie

UserContext a fonctionné comme un composant dans App.js, mais ici vous l'utilisez davantage comme un élément de données. Cependant, il peut toujours servir de composant si vous le souhaitez. Vous pouvez accéder aux mêmes données en utilisant un Consumer qui fait partie du UserContext. Vous récupérez les données en ajoutant UserContext.Consumer à votre JSX, puis utilisez une [fonction en tant qu'enfant pour accéder aux données](https://reactjs.org/docs/context.html#contextconsumer) .

Bien qu'il soit possible d'utiliser le composant Consumer, l'utilisation de Hooks peut souvent être plus courte et plus facile à lire, tout en fournissant les mêmes informations à jour. C'est pourquoi ce tutoriel utilise l'approche Hooks.

Enregistrez et fermez le fichier. Lorsque vous le faites, la page se rafraîchira et vous verrez le même nom. Mais cette fois, il s'est mis à jour dynamiquement :



Dans ce cas, les données n'ont pas voyagé à travers de nombreux composants. L'arborescence des composants qui représente le chemin parcouru par les données ressemblerait à ceci :

| UserContext.Provider

| Navigation

Vous pouvez transmettre ce nom d'utilisateur comme accessoire, et à cette échelle, cela pourrait être une stratégie efficace. Mais au fur et à mesure que l'application se développe, il est possible que le composant Navigation se déplace. Il peut y avoir un composant appelé Header qui encapsule le composant Navigation et un autre composant tel qu'un TitleBar, ou peut-être allez-vous créer un composant Template puis imbriquer le Navigation dedans. En utilisant le contexte, vous n'aurez pas à refactoriser Navigation tant que le Provider sera dans l'arborescence, ce qui facilitera la refactorisation.

Le composant suivant qui a besoin de données utilisateur est le composant SaladItem. Dans le composant SaladItem, vous aurez besoin du tableau de favoris de l'utilisateur. Vous afficherez conditionnellement l'emoji si l'ingrédient est un favori de l'utilisateur.

Ouvrez SaladItem.js:

1. nano src/components/SaladItem/SaladItem.js

Copie

Importez useContext et UserContext, puis appelez useContext avec UserContext. Après cela, vérifiez si l'ingrédient est dans le favorites tableau en utilisant la méthode  includes:

état-contexte-tutoriel/src/components/SaladItem/SaladItem.js

import React, { useContext } from 'react';

import PropTypes from 'prop-types';

import { createUseStyles } from 'react-jss';

import UserContext from '../User/User';

const useStyles = createUseStyles({

...

});

export default function SaladItem({ image, name }) {

const classes = useStyles();

const user = useContext(UserContext);

const favorite = user.favorites.includes(name);

return(

<div className={classes.wrapper}>

<h3>

{name}

</h3>

<span className={classes.favorite} aria-label={favorite ? 'Favorite' : 'Not Favorite'}>

{favorite ? '😋' : ''}

</span>

<button className={classes.add}>

<span className={classes.image} role="img" aria-label={name}>{image}</span>

</button>

</div>

)

}

SaladItem.propTypes = {

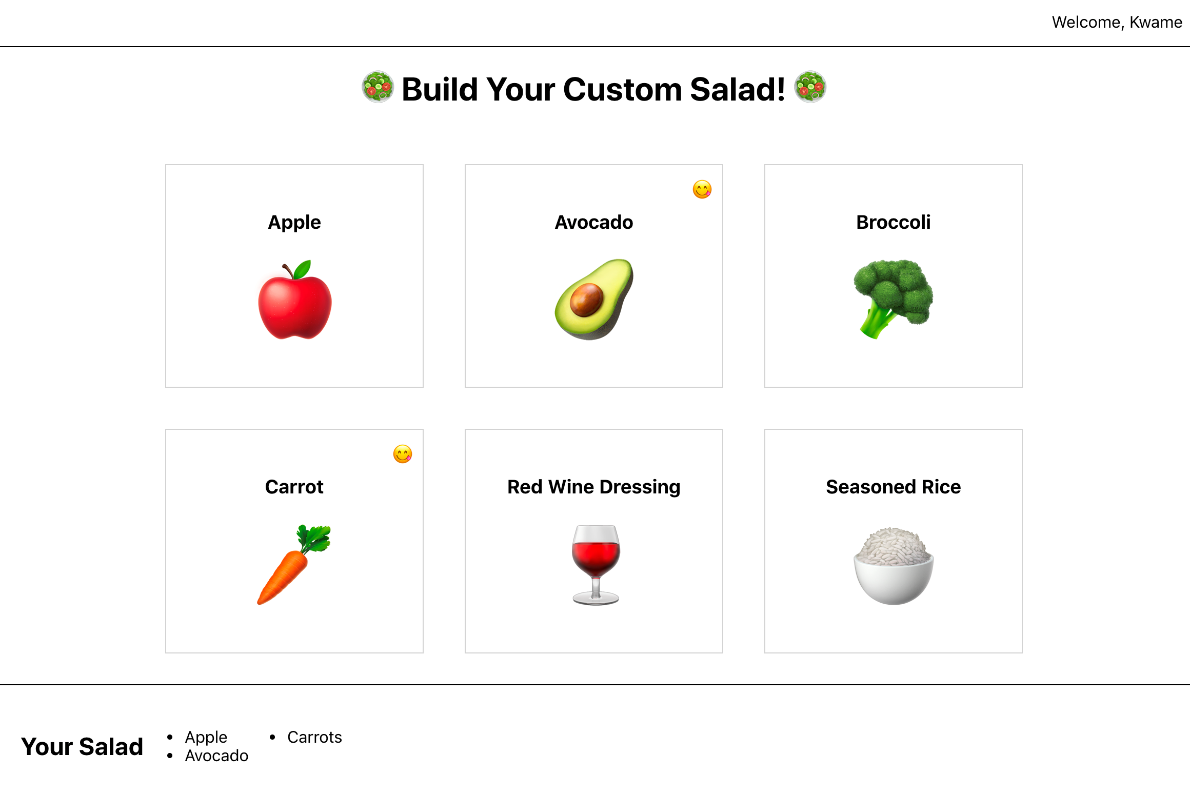
image: PropTypes.string.isRequired,

name: PropTypes.string.isRequired,

}

Copie

Enregistrez et fermez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et vous verrez que seuls les éléments favoris ont l'emoji :



Contrairement à Navigation, le contexte voyage beaucoup plus loin. L'arborescence des composants ressemblerait à ceci :

| User.Provider

| SaladMaker

| SaladBuilder

| SaladItem

Les informations ont sauté deux composants intermédiaires sans aucune prop. Si vous deviez transmettre les données en tant que prop tout au long de l'arborescence, cela représenterait beaucoup de travail et vous risqueriez qu'un futur développeur refactorise le code et oublie de transmettre la prop. Avec le contexte, vous pouvez être sûr que le code fonctionnera à mesure que l'application grandit et évolue.

Dans cette étape, vous avez créé un contexte et utilisé un Provider pour définir les données dans l'arborescence des composants. Vous avez également accédé au contexte avec le hook useContext et utilisé le contexte sur plusieurs composants. Ces données étaient statiques et n'ont donc jamais changé après la configuration initiale, mais il y aura des moments où vous devrez partager des données et également modifier les données sur plusieurs composants. À l'étape suivante, vous mettrez à jour les données imbriquées à l'aide du contexte.

**Étape 3 - Mise à jour des données à partir de composants imbriqués**

Dans cette étape, vous utiliserez le contexte et le hook useReducer pour créer des données dynamiques que les composants imbriqués peuvent consommer et mettre à jour. Vous mettrez à jour vos composants SaladItem pour définir les données qui seront utilisées et affichées pa SaladSummary. Vous définirez également des fournisseurs de contexte en dehors du composant racine. À la fin de cette étape, vous disposerez d'une application capable d'utiliser et de mettre à jour des données sur plusieurs composants et vous pourrez ajouter plusieurs fournisseurs de contexte à différents niveaux d'une application.

À ce stade, votre application affiche des données utilisateur sur plusieurs composants, mais il n'y a aucune interaction avec l'utilisateur. À l'étape précédente, vous avez utilisé le contexte pour partager une seule donnée, mais vous pouvez également partager une collection de données, y compris des fonctions. Cela signifie que vous pouvez partager des données et également partager la fonction de mise à jour des données.

Dans votre application, chacun des SaladItem doit mettre à jour une liste partagée. Ensuite, votre composant SaladSummary affichera les éléments que l'utilisateur a sélectionnés et les ajoutera à la liste. Le problème est que ces composants ne sont pas des descendants directs, vous ne pouvez donc pas transmettre les données et les fonctions de mise à jour en tant que Props. Mais ils partagent un parent commun : SaladMaker.

L'une des grandes différences entre le contexte et d'autres solutions de gestion d'état telles que Redux est que le contexte n'est pas destiné à être un magasin central. Vous pouvez l'utiliser plusieurs fois dans une application et l'initier au niveau racine ou profondément dans une arborescence de composants. En d'autres termes, vous pouvez répartir vos contextes dans l'ensemble de l'application, en créant des collectes de données ciblées sans vous soucier des conflits.

Pour rester centré sur le contexte, créez Providers cette enveloppe du parent partagé le plus proche lorsque cela est possible. Dans ce cas, cela signifie que plutôt que d'ajouter un autre contexte dans App, vous ajouterez le contexte dans le composant SaladMaker.

Ouvrez SaladMaker :

1. nano src/components/SaladMaker/SaladMaker.js

Copie

Créez et exportez ensuite un nouveau contexte nommé SaladContext :

état-contexte-tutoriel/src/components/SaladMaker/SaladMaker.js

import React, { createContext } from 'react';

import { createUseStyles } from 'react-jss';

import SaladBuilder from '../SaladBuilder/SaladBuilder';

import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({

wrapper: {

textAlign: 'center',

}

});

export const SaladContext = createContext();

export default function SaladMaker() {

const classes = useStyles();

return(

<>

<h1 className={classes.wrapper}>

<span role="img" aria-label="salad">🥗 </span>

Build Your Custom Salad!

<span role="img" aria-label="salad"> 🥗</span>

</h1>

<SaladBuilder />

<SaladSummary />

</>

)

}

Copie

À l'étape précédente, vous avez créé un composant distinct pour votre contexte, mais dans ce cas, vous le créez dans le même fichier que vous l'utilisez. Étant donné que User ne semble pas directement lié au App, il serait peut-être plus judicieux de les séparer. Cependant, étant donné que le SaladContext est étroitement lié au composant SaladMaker, les garder ensemble créera un code plus lisible.

En outre, vous pouvez créer un contexte plus générique appelé OrderContext, que vous pouvez réutiliser dans plusieurs composants. Dans ce cas, vous voudriez créer un composant séparé. Pour l'instant, gardez-les ensemble. Vous pouvez toujours refactoriser plus tard si vous décidez de passer à un autre modèle.

Avant d'ajouter la réflexion Provider sur les données que vous souhaitez partager. Vous aurez besoin d'un tableau d'éléments et d'une fonction pour ajouter les éléments. Contrairement à d'autres outils de gestion d'état centralisée, le contexte ne gère pas les mises à jour de vos données. Il contient simplement les données pour une utilisation ultérieure. Pour mettre à jour les données, vous devrez utiliser d'autres outils de gestion d'état tels que Hooks. Si vous collectiez des données pour le même composant, vous utiliseriez les hooks useState ou useReducer. Si vous débutez avec ces Hooks, consultez [How To Manage State with Hooks on React Components](https://www.digitalocean.com/community/tutorials/how-to-manage-state-with-hooks-on-react-components) .

Le hook useReducer est un bon choix car vous devrez mettre à jour l'état le plus récent à chaque action.

Créez une fonction reducer qui ajoute un nouvel élément à un tableau state, puis utilisez le hook useReducer pour créer un tableau salad et une fonction setSalad:

état-contexte-tutoriel/src/components/SaladMaker/SaladMaker.js

import React, { useReducer, createContext } from 'react';

import { createUseStyles } from 'react-jss';

import SaladBuilder from '../SaladBuilder/SaladBuilder';

import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({

wrapper: {

textAlign: 'center',

}

});

export const SaladContext = createContext();

function reducer(state, item) {

return [...state, item]

}

export default function SaladMaker() {

const classes = useStyles();

const [salad, setSalad] = useReducer(reducer, []);

return(

<>

<h1 className={classes.wrapper}>

<span role="img" aria-label="salad">🥗 </span>

Build Your Custom Salad!

<span role="img" aria-label="salad"> 🥗</span>

</h1>

<SaladBuilder />

<SaladSummary />

</>

)

}

Copie

Vous avez maintenant un composant qui contient les données salad que vous souhaitez partager, une fonction appelée setSalad pour mettre à jour les données et le SaladContext pour partager les données dans le même composant. À ce stade, vous devez les combiner ensemble.

Pour combiner, vous devrez créer un fichier Provider. Le problème est que le Provider prend une valeur unique value comme prop. Comme vous ne pouvez pas passer salad et setSalad individuellement, vous devrez les combiner en un objet et passer l'objet en tant que value :

état-contexte-tutoriel/src/components/SaladMaker/SaladMaker.js

import React, { useReducer, createContext } from 'react';

import { createUseStyles } from 'react-jss';

import SaladBuilder from '../SaladBuilder/SaladBuilder';

import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({

wrapper: {

textAlign: 'center',

}

});

export const SaladContext = createContext();

function reducer(state, item) {

return [...state, item]

}

export default function SaladMaker() {

const classes = useStyles();

const [salad, setSalad] = useReducer(reducer, []);

return(

<SaladContext.Provider value={{ salad, setSalad }}>

<h1 className={classes.wrapper}>

<span role="img" aria-label="salad">🥗 </span>

Build Your Custom Salad!

<span role="img" aria-label="salad"> 🥗</span>

</h1>

<SaladBuilder />

<SaladSummary />

</SaladContext.Provider>

)

}

Copie

Enregistrez et fermez le fichier. Comme pour Navigation, il peut sembler inutile de créer un contexte lorsque SaladSummary se trouve dans le même composant que le contexte. Passer salad comme prop est parfaitement raisonnable, mais vous risquez de le refactoriser plus tard. L'utilisation du contexte permet de regrouper les informations en un seul endroit.

Ensuite, allez dans le composant SaladItem et extrayez la fonction setSalad du contexte.

Ouvrez le composant dans un éditeur de texte :

1. nano src/components/SaladItem/SaladItem.js

Copie

À l' intérieur de SaladItem, importez le contexte de SaladMaker, puis extrayez la fonction setSalad à l'aide de la déstructuration. Ajoutez un événement click au bouton qui appellera la fonction setSalad. Étant donné que vous souhaitez qu'un utilisateur puisse ajouter un élément plusieurs fois, vous devrez également créer un identifiant unique pour chaque élément afin que la fonction map puisse attribuer un unique key :

état-contexte-tutoriel/src/components/SaladItem/SaladItem.js

import React, { useReducer, useContext } from 'react';

import PropTypes from 'prop-types';

import { createUseStyles } from 'react-jss';

import UserContext from '../User/User';

import { SaladContext } from '../SaladMaker/SaladMaker';

const useStyles = createUseStyles({

...

});

const reducer = key => key + 1;

export default function SaladItem({ image, name }) {

const classes = useStyles();

const { setSalad } = useContext(SaladContext)

const user = useContext(UserContext);

const favorite = user.favorites.includes(name);

const [id, updateId] = useReducer(reducer, 0);

function update() {

setSalad({

name,

id: `${name}-${id}`

})

updateId();

};

return(

<div className={classes.wrapper}>

<h3>

{name}

</h3>

<span className={classes.favorite} aria-label={favorite ? 'Favorite' : 'Not Favorite'}>

{favorite ? '😋' : ''}

</span>

<button className={classes.add} onClick={update}>

<span className={classes.image} role="img" aria-label={name}>{image}</span>

</button>

</div>

)

}

...

Copie

Pour rendre l'identifiant unique, vous utiliserez le crochet useReducer pour incrémenter une valeur à chaque clic. Pour le premier clic, l'identifiant sera 0 ; le second sera 1, et ainsi de suite. Vous n'afficherez jamais cette valeur à l'utilisateur ; cela créera simplement une valeur unique pour la fonction de mappage plus tard.

Après avoir créé l'identifiant unique, vous avez créé une fonction appelée update pour incrémenter l'identifiant et appeler setSalad. Enfin, vous avez attaché la fonction au bouton avec la prop onClick.

Enregistrez et fermez le fichier. La dernière étape consiste à extraire les données dynamiques du contexte dans le fichier SaladSummary.

Ouvrez SaladSummary :

1. nano src/components/SaladSummary/SaladSummary.js

Copie

Importez le composant SaladContext, puis extrayez les données salad à l'aide de la [déstructuration](https://www.digitalocean.com/community/tutorials/understanding-destructuring-rest-parameters-and-spread-syntax-in-javascript) . Remplacez les éléments de liste codés en dur par une fonction qui mappe sur salad, convertissant les objets en éléments <li>. Assurez-vous d'utiliser le id comme key :

état-contexte-tutoriel/src/components/SaladSummary/SaladSummary.js

import React, { useContext } from 'react';

import { createUseStyles } from 'react-jss';

import { SaladContext } from '../SaladMaker/SaladMaker';

const useStyles = createUseStyles({

...

});

export default function SaladSummary() {

const classes = useStyles();

const { salad } = useContext(SaladContext);

return(

<div className={classes.wrapper}>

<h2>Your Salad</h2>

<ul className={classes.list}>

{salad.map(({ name, id }) => (<li key={id}>{name}</li>))}

</ul>

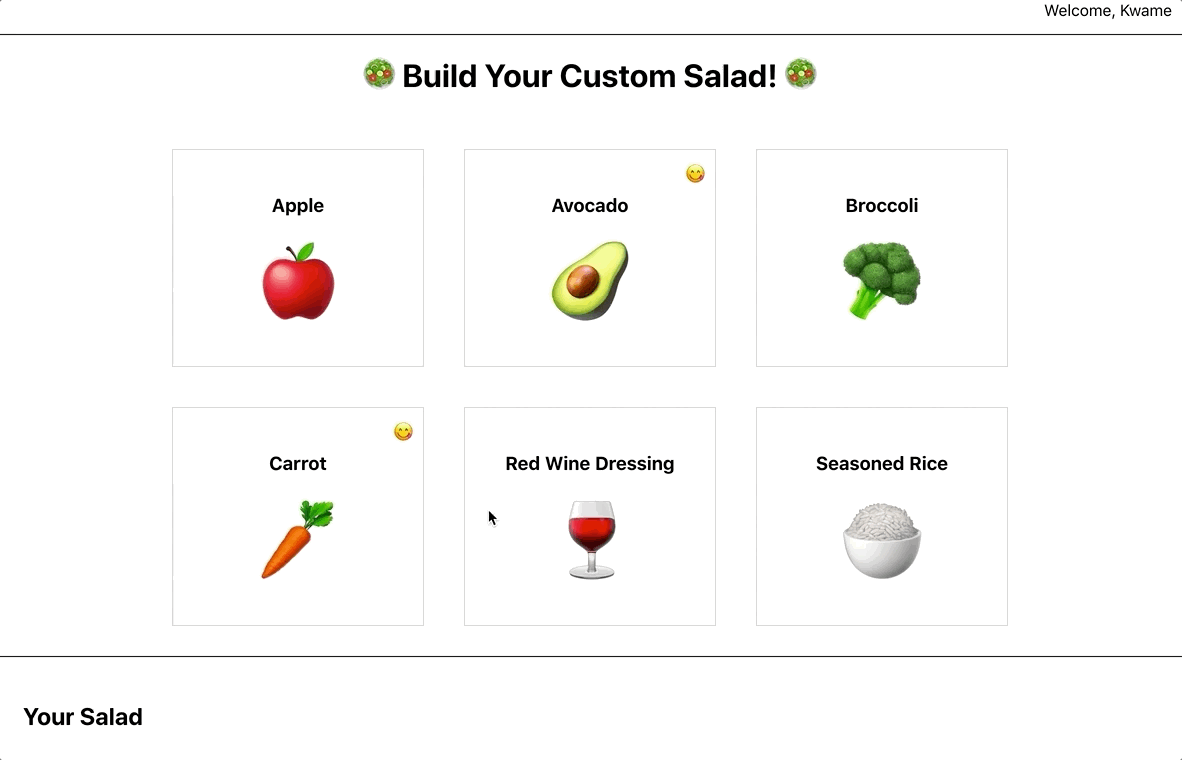
</div>

)

}

Copie

Enregistrez et fermez le fichier. Lorsque vous le ferez, vous pourrez cliquer sur les éléments et cela mettra à jour le résumé :



Remarquez comment le contexte vous a donné la possibilité de partager et de mettre à jour des données dans différents composants. Le contexte n'a pas mis à jour les éléments eux-mêmes, mais il vous a permis d'utiliser le hook useReducer sur plusieurs composants. De plus, vous aviez également la liberté de placer le contexte plus bas dans l'arborescence. Il peut sembler préférable de toujours garder le contexte à la racine, mais en gardant le contexte plus bas, vous n'avez pas à vous soucier de l'état inutilisé qui reste dans un magasin central. Dès que vous démontez un composant, les données disparaissent. Cela peut poser problème si jamais vous souhaitez enregistrer les données, mais dans ce cas, il vous suffit d'élever le contexte à un parent supérieur.

Un autre avantage d'utiliser le contexte plus bas dans votre arborescence d'application est que vous pouvez réutiliser un contexte sans vous soucier des conflits. Supposons que vous disposiez d'une application plus grande comprenant une machine à sandwich et une machine à salade. Vous pouvez créer un contexte générique appelé OrderContext, puis l'utiliser à plusieurs endroits de votre composant sans vous soucier des conflits de données ou de noms. Si vous aviez un SaladMaker et un SandwichMaker, l'arbre ressemblerait à ceci :

| App

| Salads

| OrderContext

| SaladMaker

| Sandwiches

| OrderContext

| SandwichMaker

Remarquez qu'il y a deux fois OrderContext. C'est bien, puisque le hook useContext cherchera le fournisseur le plus proche.

Au cours de cette étape, vous avez partagé et mis à jour des données à l'aide du contexte. Vous avez également placé le contexte en dehors de l'élément racine afin qu'il soit proche des composants qui ont besoin des informations sans encombrer un composant racine. Enfin, vous avez combiné le contexte avec les hooks de gestion d'état pour créer des données dynamiques et accessibles sur plusieurs composants.

**Conclusion**

Context est un outil puissant et flexible qui vous permet de stocker et d'utiliser des données dans une application. Il vous donne la possibilité de gérer des données distribuées avec des outils intégrés qui ne nécessitent aucune installation ou configuration tierce supplémentaire.

La création de contextes réutilisables est importante pour une variété de composants communs tels que les formulaires qui doivent accéder aux données à travers les éléments ou les onglets qui nécessitent un contexte commun pour l'onglet et l'affichage. Vous pouvez stocker de nombreux types d'informations dans des contextes, notamment des thèmes, des données de formulaire, des messages d'alerte, etc. Le contexte vous donne la liberté de créer des composants qui peuvent accéder aux données sans vous soucier de la façon de transmettre les données via des composants intermédiaires ou de stocker les données dans un magasin centralisé sans rendre le magasin trop volumineux.