**Comment styliser les composants React**

introduction

Dans ce didacticiel, vous apprendrez trois façons différentes de [styliser les](https://reactjs.org/) composants [React](https://reactjs.org/) : des [feuilles de style en cascade (CSS)](https://developer.mozilla.org/en-US/docs/Web/CSS) simples , des styles en ligne avec [des objets de style JavaScript](https://www.digitalocean.com/community/tutorials/understanding-objects-in-javascript) et [JSS](https://cssinjs.org/) , une bibliothèque de création de CSS avec [JavaScript](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-javascript) . Ces options présentent chacune des avantages et des inconvénients, certaines vous offrant une meilleure protection contre les conflits de style ou vous permettant de vous référer directement à des accessoires ou à d'autres données dynamiques. Mais toutes les options ont une chose en commun : elles vous permettent de conserver les styles spécifiques à vos composants à proximité du composant, ce qui facilite la réutilisation des composants dans un projet ou même dans de nombreux projets non liés.

Chacune de ces options repose sur les propriétés CSS. Pour utiliser du CSS brut sans aucune donnée d'exécution, vous pouvez importer des feuilles de style. Si vous souhaitez créer des styles intégrés au composant, vous pouvez utiliser des objets de style en ligne qui utilisent des noms de propriété CSS comme clés et le style comme valeur. Enfin, si vous souhaitez une combinaison, vous pouvez utiliser une bibliothèque tierce telle que JSS pour écrire votre CSS dans la syntaxe JavaScript, un concept logiciel appelé *CSS-in-JS*.

Pour illustrer ces méthodes, vous allez créer un exemple de composant alert qui affichera soit un style de réussite, soit un style d'erreur en fonction de l' [accessoire](https://www.digitalocean.com/community/tutorials/how-to-customize-react-components-with-props) . Le composant alert prendra n'importe quel nombre d'enfants. Cela signifie que vous devrez faire attention aux conflits de style, car vous n'avez aucun moyen de savoir quels styles auront les composants enfants. Après avoir créé le composant alert, vous le refactoriserez en utilisant chacune des options de style afin que vous puissiez voir les similitudes et les différences entre les approches.

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 décrites dans [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 dans 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) .
* Vous devrez être en mesure de créer des applications avec [Create React App](https://github.com/facebook/create-react-app) . Vous pouvez trouver des instructions pour installer une application avec Create React App sur [Comment configurer un projet React avec Create React App](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-create-react-app) .
* Vous utiliserez des composants React, que vous pouvez découvrir dans notre tutoriel [Comment créer des composants personnalisés dans React](https://www.digitalocean.com/community/tutorials/how-to-create-custom-components-in-react) .
* Vous aurez également besoin d'une connaissance de base de JavaScript, que vous pouvez trouver dans la série [Comment coder en 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 bonne ressource pour HTML et CSS est le [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML) .

Étape 1 - Création d'un projet vide

Dans cette étape, vous allez créer un nouveau projet à l'aide de [Create React App](https://github.com/facebook/create-react-app) . Ensuite, vous supprimerez l'exemple de projet et les fichiers associés qui sont installés lorsque vous amorcez le projet. Enfin, vous allez créer une structure de fichiers simple pour organiser vos composants. Cela vous donnera une base solide sur laquelle créer l'exemple d'application de ce didacticiel pour le style à l'étape suivante.

Pour commencer, créez un nouveau projet. Dans votre terminal, exécutez le script suivant pour installer un nouveau projet en utilisant create-react-app:

* npx create-react-app styling-tutorial

Copie

Une fois le projet terminé, accédez au répertoire:

* cd styling-tutorial

Copie

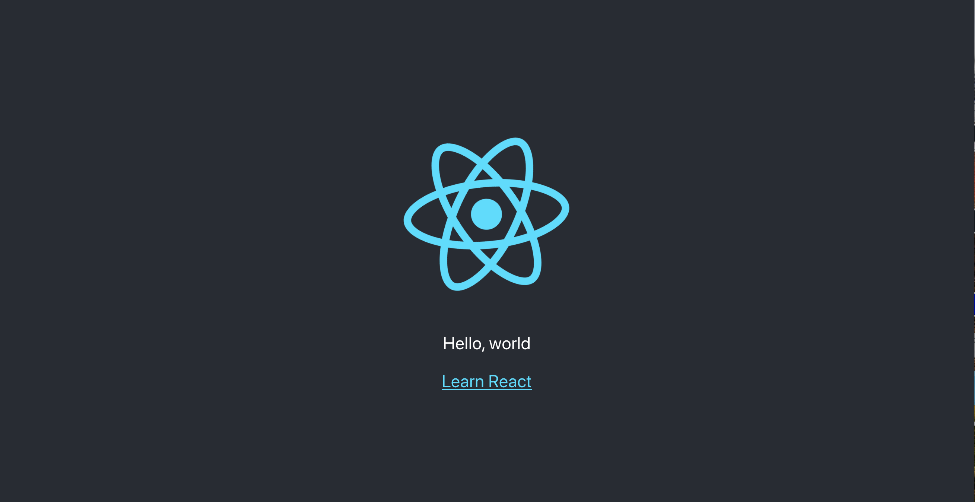
Dans un nouvel onglet ou une nouvelle fenêtre de terminal, démarrez le projet à l'aide du [script de démarrage Create React App](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-create-react-app#step-3-%E2%80%94-starting-the-server) . Le navigateur s'actualisera automatiquement en cas de modification, alors laissez ce script en cours d'exécution pendant que vous travaillez:

* npm start

Copie

Vous obtiendrez un serveur local en cours d'exécution. Si le projet ne s'est pas ouvert dans une fenêtre de navigateur, vous pouvez l'ouvrir avec <http://localhost:3000/>. Si vous l'exécutez à partir d'un serveur distant, l'adresse sera .http://your\_domain:3000

Votre navigateur se chargera avec une simple application React incluse dans le cadre de Create React App:



Vous allez créer un tout nouvel ensemble de composants personnalisés, vous devrez donc commencer par effacer du code standard pour pouvoir avoir un projet vide.

Pour commencer, ouvrez src/App.jsdans un éditeur de texte. C'est le composant racine qui est injecté dans la page. Tous les composants commenceront à partir d'ici. Vous pouvez trouver plus d' informations sur App.jsà [Comment mettre en place un projet REACT avec App Créer React](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-create-react-app#step-5-%E2%80%94-modifying-the-heading-tag-and-styling) .

Ouvrez src/App.jsavec la commande suivante:

* nano src/App.js

Copie

Vous verrez un fichier comme celui-ci:

styling-tutorial / src / App.js

import React from 'react';

import logo from './logo.svg';

import './App.css';

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Edit <code>src/App.js</code> and save to reload.

</p>

<a

className="App-link"

href="https://reactjs.org"

target="\_blank"

rel="noopener noreferrer"

>

Learn React

</a>

</header>

</div>

);

}

export default App;

Copie

Supprimez la ligne import logo from './logo.svg';. Remplacez ensuite tout dans l' returninstruction pour renvoyer un ensemble de balises vides: <></>. Cela vous donnera une page valide qui ne renvoie rien. Le code final ressemblera à ceci:

styling-tutorial / src / App.js

import React from 'react';

import './App.css';

function App() {

return <></>;

}

export default App;

Copie

Enregistrez et quittez l'éditeur de texte.

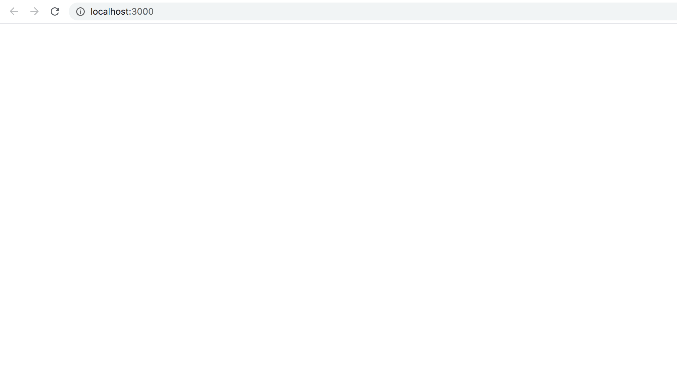
Enfin, supprimez le logo. Vous ne l'utiliserez pas dans votre application et vous devez supprimer les fichiers inutilisés pendant que vous travaillez. Cela vous évitera la confusion à long terme.

Dans la fenêtre du terminal, tapez la commande suivante:

* rm src/logo.svg

Copie

Si vous regardez votre navigateur, vous verrez un écran vide.



Maintenant que vous avez effacé l'exemple de projet Create React App, créez une structure de fichiers simple. Cela vous aidera à garder vos composants isolés et indépendants.

Créez un répertoire appelé componentsdans le srcrépertoire. Cela contiendra tous vos composants personnalisés.

* mkdir src/components

Copie

Chaque composant aura son propre répertoire pour stocker le fichier du composant avec les styles, images et tests.

Créez un répertoire pour App:

* mkdir src/components/App

Copie

Déplacez tous les fichiers App dans ce répertoire. Utilisez le caractère générique, \* pour sélectionner les fichiers commençant par, App quelle que soit l'extension de fichier. Ensuite, utilisez la mv commande pour les mettre dans le nouveau répertoire:

* mv src/App.\* src/components/App

Copie

Ensuite, mettez à jour le chemin d'importation relatif dans index.js, qui est le composant racine qui amorce l'ensemble du processus :

* nano src/index.js

Copie

L'instruction d'importation doit pointer vers le fichier App.js dans le répertoire App, apportez donc la modification suivante en surbrillance:

styling-tutorial / src / index.js

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './components/App/App';

import \* as serviceWorker from './serviceWorker';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

Copie

Enregistrez et quittez le fichier.

Maintenant que le projet est configuré, vous pouvez créer votre premier composant.

Étape 2 - Styliser un composant avec du CSS simple

Dans cette étape, vous allez créer un exemple de composant Alert qui affichera une alerte sur une page Web. Vous allez le styliser en utilisant du CSS simple, que vous importerez directement dans le composant. Cela garantira que les styles du composant restent étroitement associés au JavaScript et au [JSX](https://www.digitalocean.com/community/tutorials/how-to-create-react-elements-with-jsx) du composant. Vous allez également créer un composant qui implémentera le composant Alert pour voir comment les styles peuvent affecter les enfants et comment vous pouvez utiliser des accessoires pour modifier les styles de manière dynamique.

À la fin de cette étape, vous aurez créé plusieurs composants utilisant du CSS brut importé directement dans le composant.

Création d'un composant d'alerte

Pour commencer, créez un nouveau composant Alert. Commencez par créer le répertoire :

* mkdir src/components/Alert

Copie

Ensuite, ouvrez Alert.js:

* nano src/components/Alert/Alert.js

Copie

Ajoutez un composant de base qui renvoie la chaîne Alert:

styling-tutorial / src / components / Alert / Alert.js

import React from 'react';

export default function Alert() {

return <div>Alert</div>

}

Copie

Enregistrez et fermez le fichier.

Ensuite, ouvrez App.js:

* nano src/components/App/App.js

Copie

Importez le composant Alert et rendez-le dans un <div>en ajoutant le code en surbrillance :

styling-tutorial / src / components / App / App.js

import React from 'react';

import './App.css';

import Alert from '../Alert/Alert';

function App() {

return (

<div className="wrapper">

<Alert />

</div>

)

}

export default App;

Copie

Dans ce code, vous avez donné a <div> le className wrapper, qui sera utilisé pour le style plus tard.

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez votre composant :



Ensuite, vous allez styliser le composant App pour lui donner un peu de remplissage afin que le composant Alert ne soit pas si près du bord. Ouvrez le fichier App.css:

* nano src/components/App/App.css

Ce fichier utilise le CSS standard. Pour ajouter un remplissage au wrapper, remplacez le code par défaut par une règle comme vous le feriez pour CSS dans un projet HTML simple. Dans ce cas, ajoutez un padding de 20px :

styling-tutorial / src / components / App / App.css

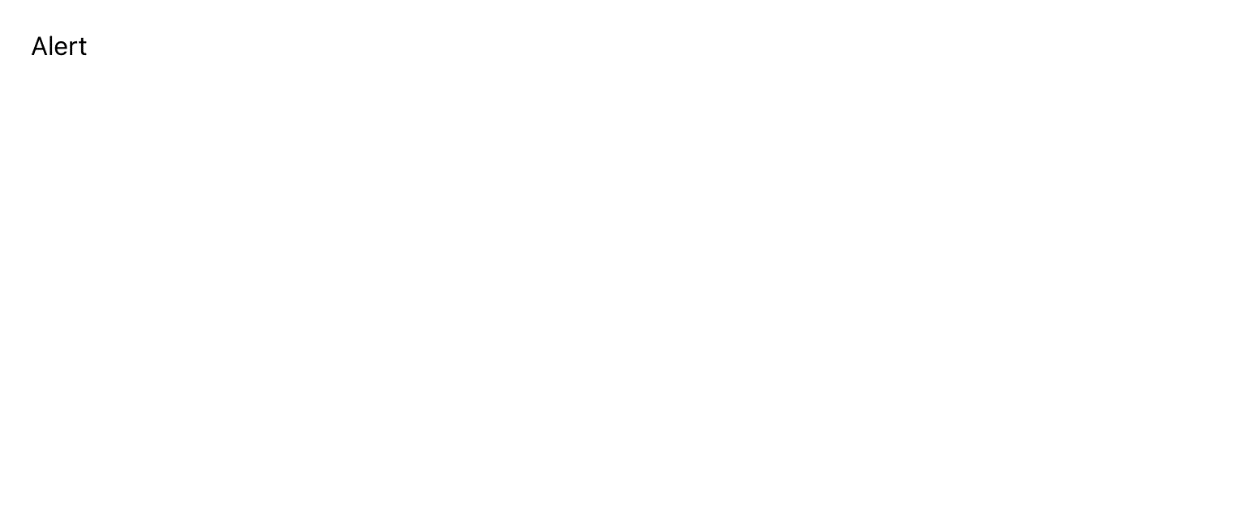
.wrapper {

padding: 20px;

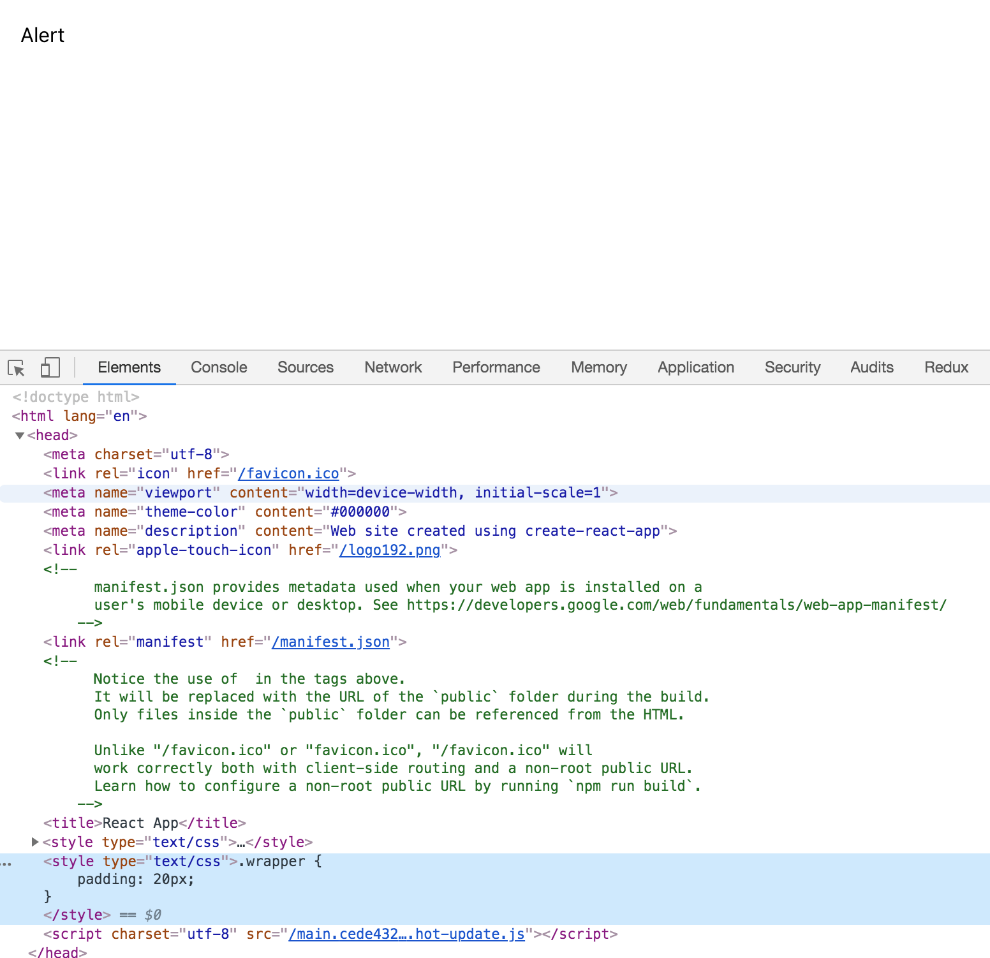
}

Copie

Enregistrez et fermez le fichier. Lorsque vous le faites, la page s'actualisera et vous verrez le remplissage supplémentaire :



Lorsque vous utilisez Create React App, [webpack](https://webpack.js.org/) prend le CSS importé et l'ajoute à une balise de style en haut du fichier rendu dans le navigateur. Si vous regardez l'élément <head> dans la source de votre page, vous verrez les styles :



Cela signifie que vous pouvez conserver le CSS avec le composant et il sera collecté pendant la [phase de construction](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-create-react-app#step-6-%E2%80%94-building-the-project) . Cela signifie également que vos styles ont une portée globale, ce qui peut créer des conflits de noms potentiels. Avec cette méthode, chaque nom de classe devra être unique dans tous les composants.

Pour explorer ce problème, vous allez apporter des modifications au Alertcomposant.

Tout d'abord, ouvrez le fichier:

* nano src/components/Alert/Alert.js

Copie

Puis ajouter un peu React code qui prendra children, typeet titlecomme accessoires:

styling-tutorial / src / components / Alert / Alert.js

import React from 'react';

import PropTypes from 'prop-types';

export default function Alert({ children, title, type }) {

return (

<div>

<h2>{title}</h2>

{children}

</div>

)

}

Alert.propTypes = {

children: PropTypes.oneOfType([

PropTypes.arrayOf(PropTypes.element),

PropTypes.element.isRequired

]),

title: PropTypes.string.isRequired,

type: PropTypes.string.isRequired,

}

Le title dans ce code est dans une balise <h2> et children vous permettra d'afficher les composants enfants. Vous utiliserez bientôt la prop type pour définir un succès et une alerte d'erreur en fonction du système de saisie [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).

Enregistrez et fermez le fichier. Ensuite, mettez à jour le composant  Alert App pour utiliser les nouvelles props.

Tout d'abord, ouvrez App.js:

* nano src/components/App/App.js

Copie

Faites une alerte qui informe un utilisateur qu'une tentative d'ajout d'articles à un panier a échoué:

styling-tutorial / src / components / App / App.js

import React from 'react';

import './App.css';

import Alert from '../Alert/Alert';

function App() {

return (

<div className="wrapper">

<Alert title="Items Not Added" type="error">

<div>Your items are out of stock.</div>

</Alert>

</div>

)

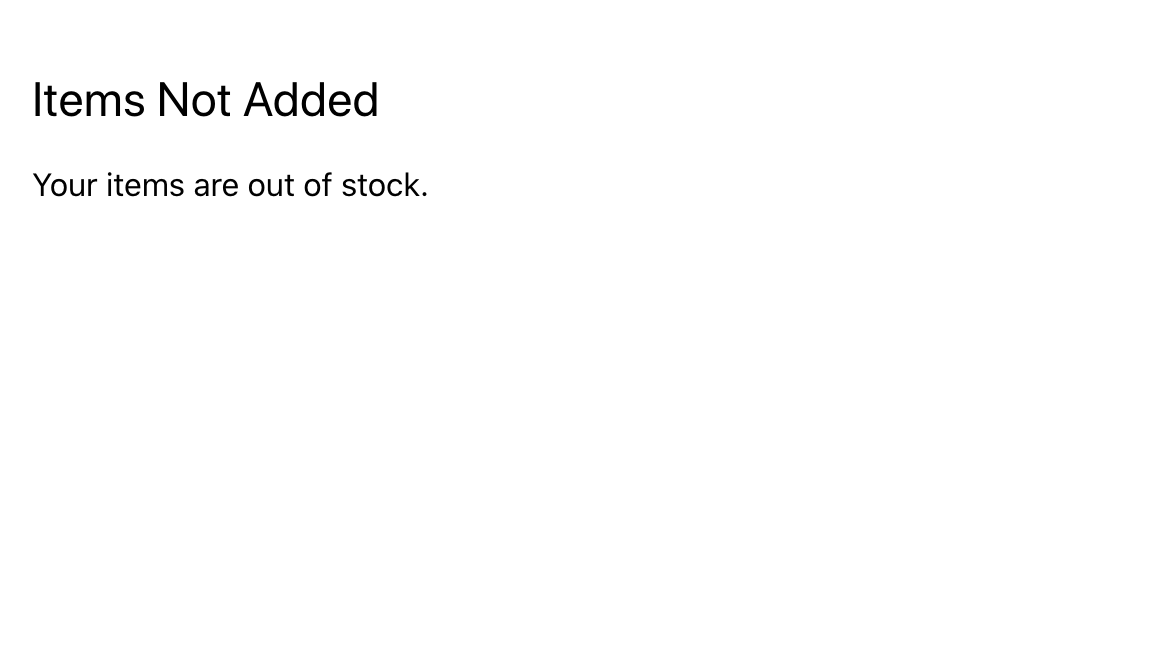
}

export default App;

Copie

Dans ce code, vous avez mis à jour le title et children avec un message d'échec, puis vous avez ajouté un type de error.

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez votre nouveau composant :



Votre alerte est en cours de rendu, donc l'étape suivante consiste à styliser le composant avec CSS.

Ajout de CSS au composant Alert

Étant donné que le composant Alert affiche une erreur, vous allez ajouter une bordure et définir la couleur de la bordure sur une nuance de rouge. Vous donnerez également à l'étiquette <h2> la même couleur. Mais cela pose un problème : vous ne pouvez pas utiliser le nom wrapper à l'extérieur de <div> de votre composant Alert, car ce nom est déjà pris par le composant App.

Les conflits de noms de classes ne sont pas un problème nouveau dans CSS, et il y a eu un certain nombre de tentatives pour les résoudre en utilisant des conventions de dénomination telles que [BEM](http://getbem.com/introduction/) . Mais les conventions de dénomination peuvent devenir verbeuses et peuvent encore occasionner des conflits dans les projets comportant un grand nombre de composants.

Plutôt que d'utiliser un ensemble spécifique de règles séparées par une convention de dénomination, dans ce didacticiel, vous préfixerez le wrapper nom de la classe avec le nom du composant. Votre nouveau nom de classe sera alert-wrapper. De plus, vous ajouterez le type de l'alerte en tant que classe.

Ouvrez le composant Alert:

* nano src/components/Alert/Alert.js

Copie

Ensuite, ajoutez le code en surbrillance suivant :

styling-tutorial / src / components / Alert / Alert.js

import React from 'react';

import PropTypes from 'prop-types';

import './Alert.css';

...

export default function Alert({ children, type, title }) {

return(

<div className={`alert-wrapper ${type}`}>

<h2>{title}</h2>

{children}

</div>

)

}

...

Copie

Dans ce cas, vous combinez alert-wrapperet la typevariable en une seule chaîne à l'aide d'un [littéral de modèle](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) .

Enregistrez et fermez le fichier. Vous avez maintenant un nom de classe unique qui change dynamiquement en fonction de l'accessoire. Le JSX dans ce code se résoudra en a div avec les noms de classe de alert-wrapperet error. La majoration compilé serait la suivante: <div class="alert-wrapper error">.

Ajoutez maintenant les styles. Tout d'abord, ouvrez le CSS du composant Alert :

* nano src/components/Alert/Alert.css

Copie

Ajouter le CSS suivant à la alert-wrapper, successet les errorclasses:

styling-tutorial / src / components / Alert / Alert.css

.alert-wrapper {

padding: 15px;

margin-bottom: 15px;

}

.alert-wrapper h2 {

margin: 0 0 10px 0;

}

.alert-wrapper.success {

border: #6DA06F solid 1px;

}

.success h2 {

color: #6DA06F;

}

.alert-wrapper.error {

border: #F56260 solid 1px;

}

.error h2 {

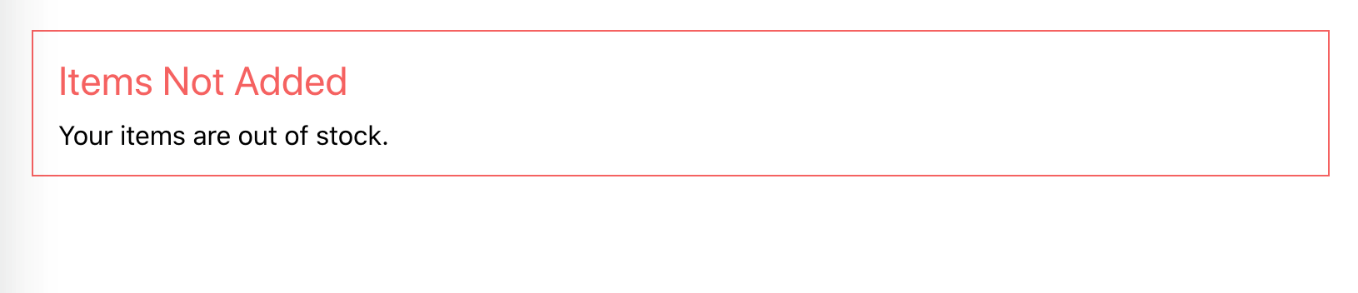
color: #F56260;

}

Copie

Ce code ajoute des marges et du remplissage au alert-wrapper. Il ajoute ensuite une bordure avec une nuance de rouge pour la classe error en utilisant le [code de couleur hexadécimal](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units#Color) #F56260 , et une nuance de vert ( #6DA06F) pour la classe success. Il met également à jour la <h2> couleur en rouge ou en vert selon le parent.

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez les nouveaux styles :



Maintenant que vous disposez d'un composant  Alert stylisé , vous pouvez créer un nouveau composant qui affiche une liste d'éléments dans le composant Alert. Étant donné que les enfants seront plus complexes, il y aura de plus grandes possibilités de conflits de style.

Création d'un composant de message de réussite

Commencez par créer un répertoire pour le nouveau composant CartSuccess:

* mkdir src/components/CartSuccess

Copie

Ouvert CartSuccess.js:

* nano src/components/CartSuccess/CartSuccess.js

Copie

À l'intérieur du composant, importez le composant Alert et passez un <div> contenant une série d'articles qu'un utilisateur a ajoutés au panier :

styling-tutorial / src / components / CartSuccess / CartSuccess.js

import React from 'react';

import Alert from '../Alert/Alert';

import './CartSuccess.css';

export default function CartSuccess() {

return(

<Alert title="Added to Cart" type="success">

<div className="cart-success-wrapper">

<h2>

You have added 3 items:

</h2>

<div className="item">

<div>Bananas</div>

<div>Quantity: 2</div>

</div>

<div className="item">

<div>Lettuce</div>

<div>Quantity: 1</div>

</div>

</div>

</Alert>

)

}

Copie

Remarquez comment vous deviez créer un nom de classe unique - cart-success-wrapper pour l'extérieur <div>. Enregistrez et fermez le fichier.

Ensuite, ajoutez du CSS au message personnalisé. Ouvert CartSuccess.css:

* nano src/components/CartSuccess/CartSuccess.css

Copie

Ajouter un [display](https://developer.mozilla.org/en-US/docs/Web/CSS/display) de [flex](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout) l'emballage. Vous voudrez que la plupart des éléments <h2>soient enveloppés, à l'exception de l'élément, qui doit prendre toute la largeur:

styling-tutorial / src / components / CartSuccess / CartSuccess.css

.cart-success-wrapper {

border-top: black solid 1px;

display: flex;

flex-wrap: wrap;

}

.cart-success-wrapper h2 {

width: 100%;

}

.item {

margin-right: 20px;

}

Copie

Ici, vous avez donné <h2> une largeur de 100%. En plus de fléchir l'élément, vous avez également ajouté une petite bordure en haut du message et ajouté une marge à la classe item pour fournir un espace entre les éléments.

Enregistrez et fermez le fichier.

Maintenant que vous disposez d'un composant stylisé, ajoutez-le à votre composant App.

Ouvert App.js:

* nano src/components/App/App.js

Copie

Importez le composant et ajoutez-le après le composant  Alert actuel , comme indiqué dans le code en surbrillance:

styling-tutorial / src / components / App / App.js

import React from 'react';

import './App.css';

import Alert from '../Alert/Alert';

import CartSuccess from '../CartSuccess/CartSuccess';

function App() {

return(

<div className="wrapper">

<Alert title="Items Not Added" type="error">

<div>Your items are out of stock.</div>

</Alert>

<CartSuccess />

</div>

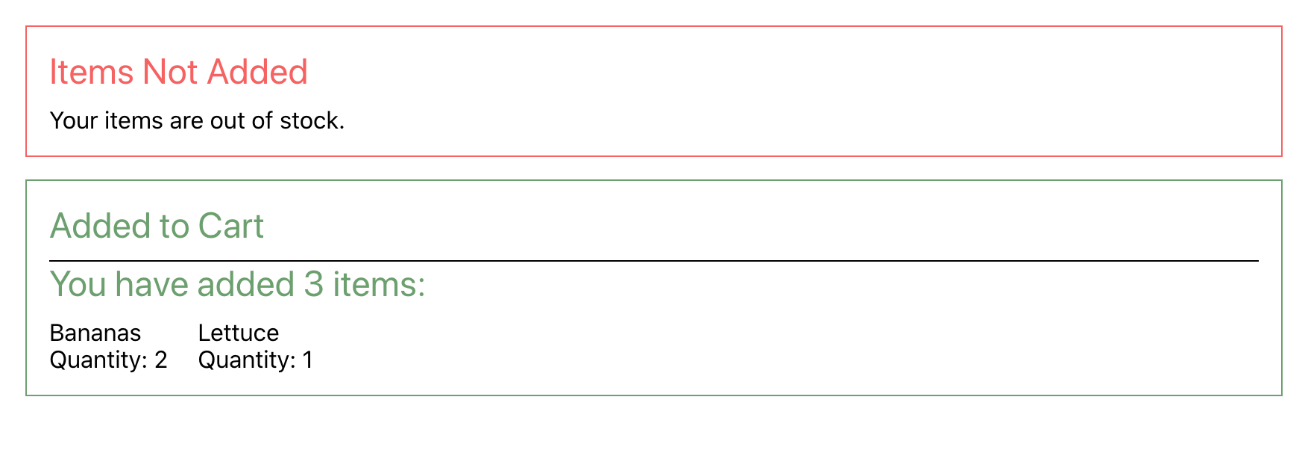
)

}

export default App;

Copie

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez votre nouveau composant :



Cela montre la nouvelle couleur et le message comme prévu, mais le composant imbriqué a reçu des styles inattendus. La règle pour le <h2>dans le composant Alert est appliquée à la <h2>balise imbriquée dans les children accessoires.

Les styles inattendus en cascade sur les enfants sont un problème courant avec CSS. Cependant, étant donné que React vous donne la possibilité de regrouper et de partager des composants entre les projets, vous avez un plus grand potentiel pour que les styles descendent par inadvertance vers les composants enfants.

Pour résoudre ce problème avec du CSS pur, rendez la <h2>règle du composant Alert un peu plus précise.

Ouvrez le fichier Alert.css:

* nano src/components/Alert/Alert.css

Copie

Modifiez les règles pour que le <h2> style s'applique uniquement aux enfants directs des classes plutôt qu'à tous les enfants à l'aide du > [combinateur enfant](https://developer.mozilla.org/en-US/docs/Web/CSS/Child_combinator) CSS :

styling-tutorial / src / components / Alert / Alert.css

.alert-wrapper {

padding: 15px;

margin-bottom: 15px;

}

.alert-wrapper > h2 {

margin: 0 0 10px 0;

}

.alert-wrapper.success {

border: #6da06f solid 1px;

}

.success > h2 {

color: #6da06f;

}

.alert-wrapper.error {

border: #f56260 solid 1px;

}

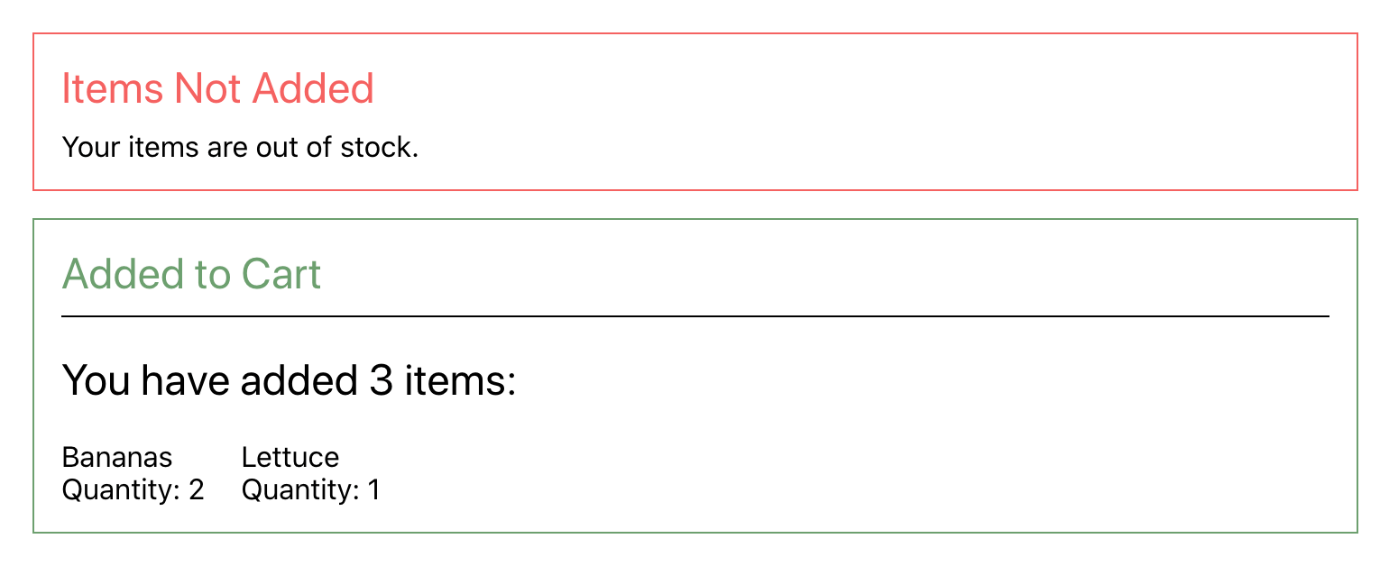
.error > h2 {

color: #f56260;

}

Copie

Enregistrez et fermez le fichier. Lorsque vous le faites, la page s'actualisera et vous verrez l' <h2>élément CartSuccessconserver la couleur par défaut:



Désormais, les styles du composant Alert n'affecteront que les enfants immédiats et ne s'appliqueront pas aux autres nœuds ou composants enfants. Cette méthode fonctionne bien dans ce cas, mais dans les circonstances où les composants sont plus complexes, il peut être difficile d'écrire des règles qui s'appliquent à tous les cas sans fuir à l'extérieur du composant.

Dans cette étape, vous avez stylisé un composant à l'aide de feuilles de style CSS importées directement dans un composant. Le style des éléments React avec du CSS standard est un moyen rapide de créer des composants avec des styles associés à l'aide de fichiers CSS standard. La facilité d'utilisation en fait une bonne première étape lorsque vous travaillez sur de nouveaux ou de petits projets, mais à mesure que les projets se développent, cela peut causer des problèmes.

Lors de la création des composants, vous avez rencontré deux problèmes de style courants: des conflits de nom de classe et une application de style involontaire. Vous pouvez les contourner avec le CSS standard, mais il existe d'autres approches de style qui vous donnent des outils pour gérer ces problèmes par programme plutôt qu'avec des conventions de dénomination. Dans l'étape suivante, vous explorerez la résolution de ces problèmes avec des objets de style.

Étape 3 - Stylisation avec des objets de style

Dans cette étape, vous allez styliser vos composants à l'aide d' *objets de style* , qui sont [des objets JavaScript](https://www.digitalocean.com/community/tutorials/understanding-objects-in-javascript) utilisant des propriétés CSS comme clés. Au fur et à mesure que vous travaillez sur vos composants, vous allez mettre à jour les clés pour qu'elles correspondent à la syntaxe JavaScript et apprendre à définir dynamiquement les propriétés de style en fonction des accessoires de composant.

Le CSS séparé est le moyen le plus courant de styliser du HTML. Cette méthode est rapide et les navigateurs sont efficaces pour appliquer des styles rapidement et de manière cohérente. Mais ce n'est pas la seule option pour styliser le HTML. En HTML standard, vous pouvez définir des styles en ligne directement sur un élément à l'aide de l' [attribut style](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/style) avec une chaîne contenant les styles que vous souhaitez appliquer.

L'une des meilleures utilisations des objets de style est le calcul dynamique des styles. Ceci est particulièrement utile si vous avez besoin de connaître la position actuelle de l'élément, car elle n'est pas déterminée tant que les éléments ne sont pas rendus et ne peuvent donc être manipulés que dynamiquement.

L'écriture manuelle de chaînes de style est difficile à faire et peut introduire des bogues. Une couleur ou un point-virgule manquant cassera la chaîne entière. Heureusement, dans JSX, vous n'êtes pas limité à une simple chaîne. L'attribut style peut également accepter un objet contenant les styles. Ces noms de style devront être [camelCase](https://en.wikipedia.org/wiki/Camel_case) plutôt que kebab-case.

Le plus grand avantage de l'utilisation de styles en ligne comme celui-ci est que, puisque vous créez des styles avec JavaScript, vous pouvez définir dynamiquement des propriétés CSS au lieu de définir dynamiquement des classes. Cela signifie que vous pouvez écrire du code sans classes CSS du tout, évitant tout conflit de nom potentiel et vous permettant de calculer les styles au moment de l'exécution.

Pour utiliser des objets de style, commencez par refactoriser App.js. Tout d'abord, ouvrez le fichier:

* nano src/components/App/App.js

Copie

A l' intérieur du composant, retirer le produit importé App.cssfichier, puis créer un objet qui a une paddingde 20et passer l'objet à l' <div>aide de l' styleattribut:

styling-tutorial / src / components / App / App.js

import React from 'react';

import Alert from '../Alert/Alert';

import CartSuccess from '../CartSuccess/CartSuccess';

function App() {

const wrapper = {

padding: 20

};

return(

<div style={wrapper}>

<Alert title="Items Not Added" type="error">

<div>Your items are out of stock.</div>

</Alert>

<CartSuccess />

</div>

)

}

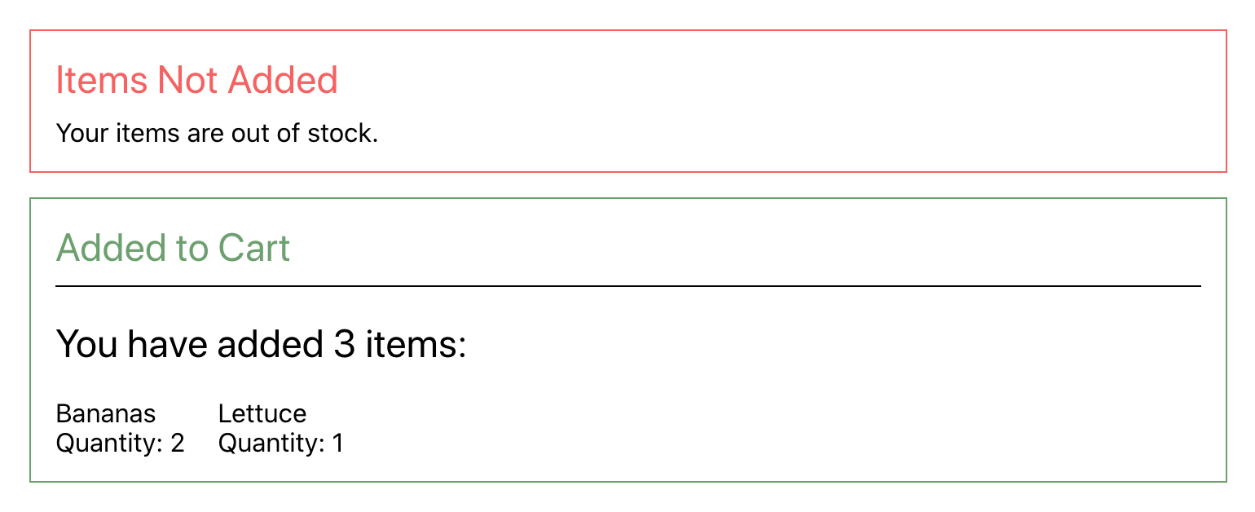
export default App;

Copie

Notez que vous ne devez pas spécifier les pixels comme unité pour padding. React le convertira en une chaîne de pixels par défaut. Si vous voulez une unité spécifique, transmettez-la sous forme de chaîne. Donc, si vous vouliez que le remplissage soit un pourcentage par exemple, ce serait le cas padding: '20%'.

La plupart des nombres seront automatiquement convertis en pixels. Il y a cependant des exceptions. La propriété line-height peut accepter des [nombres simples sans unité](https://developer.mozilla.org/en-US/docs/Web/CSS/line-height#Syntax) . Si vous souhaitez utiliser l'unité de pixels dans ce cas, vous devez spécifier les pixels sous forme de chaîne.

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez la page telle qu'elle était avant:



Ensuite, refactorisez CartSuccess. Tout d'abord, ouvrez le fichier:

* nano src/components/CartSuccess/CartSuccess.js

Copie

Comme avec App.js, supprimez le CSS importé ( CartSuccess.css) et créez un objet de style pour chaque élément qui avait auparavant une classe:

styling-tutorial / src / components / CartSuccess / CartSuccess.js

import React from 'react';

import Alert from '../Alert/Alert';

export default function CartSuccess() {

const styles = {

header: {

width: '100%'

},

item: {

marginRight: 20

},

wrapper: {

borderTop: 'black solid 1px',

display: 'flex',

flexWrap: 'wrap'

}

}

return(

<Alert title="Added to Cart" type="success">

<div style={styles.wrapper}>

<h2 style={styles.header}>

You have added 3 items:

</h2>

<div style={styles.item}>

<div>Bananas</div>

<div>Quantity: 2</div>

</div>

<div style={styles.item}>

<div>Lettuce</div>

<div>Quantity: 1</div>

</div>

</div>

</Alert>

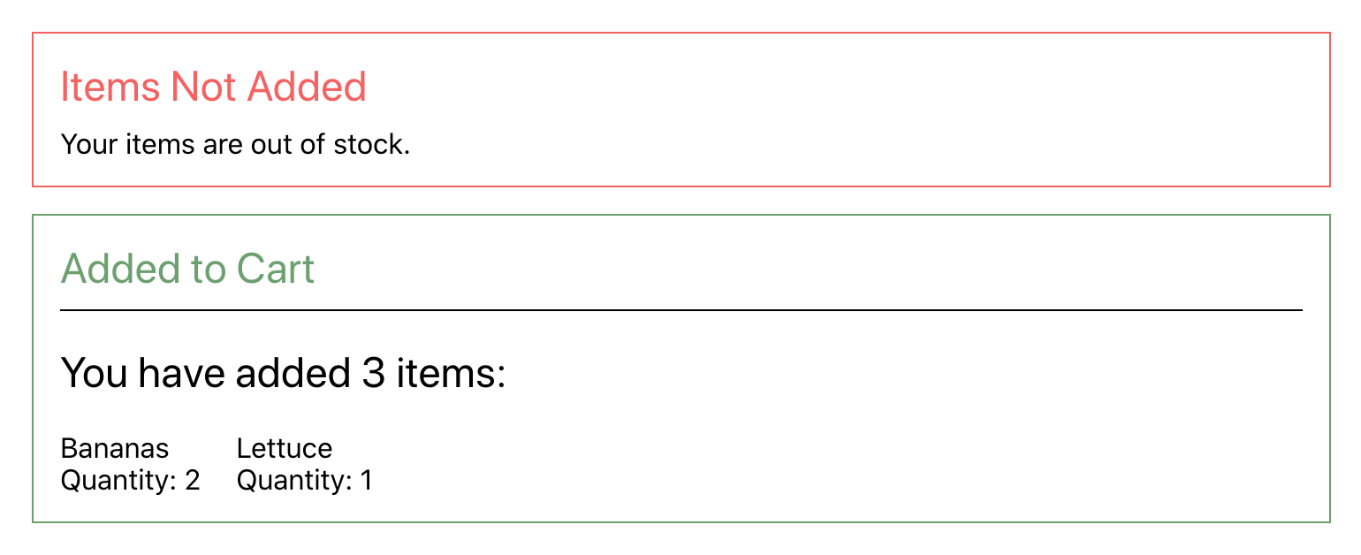
)

}

Copie

Dans ce cas, vous n'avez pas créé plusieurs objets séparés; à la place, vous avez créé un seul objet contenant d'autres objets. Notez également que vous avez besoin d'utiliser le cas de chameau pour les propriétés margin-right, border-topet flex-wrap.

Enregistrez et fermez le fichier. Lorsque vous le faites, la page s'actualisera et vous verrez la page avec les mêmes styles:



Puisque vous n'utilisez pas de classes, vous n'avez pas à vous soucier des conflits de noms. Un autre avantage de la création de styles avec JavaScript est que vous pouvez tirer parti de toute syntaxe JavaScript telle que les variables et les littéraux de modèle. Avec le CSS moderne, vous pouvez utiliser des [variables](https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties) , ce qui est une amélioration majeure, mais peut ne pas être entièrement disponible en fonction des exigences de prise en charge de votre navigateur. En particulier, ils ne sont pris en charge dans aucune version d'Internet Explorer, bien que vous puissiez utiliser un [polyfill](https://en.wikipedia.org/wiki/Polyfill_(programming)) pour ajouter une prise en charge.

Étant donné que les objets de style sont créés au moment de l'exécution, ils sont plus prévisibles et peuvent utiliser n'importe quel JavaScript pris en charge.

Pour voir comment les objets de style peuvent aider dans cette situation, refactorisez Alert.jspour utiliser des objets de style. Tout d'abord, ouvrez Alert.js:

* nano src/components/Alert/Alert.js

Copie

À l'intérieur Alert.js, supprimez import './Alert.css';et créez un objet appelé colorsqui a une propriété pour la couleur d'erreur et une propriété pour la couleur de réussite. Ensuite, convertissez le CSS en objet JavaScript à l'aide de l' typeaccessoire pour définir dynamiquement la couleur:

styling-tutorial / src / components / Alert / Alert.js

import React from 'react';

import PropTypes from 'prop-types';

export default function Alert({ children, type, title }) {

const colors = {

success: '#6da06f',

error: '#f56260',

}

const style = {

heading: {

color: colors[type],

margin: '0 0 10px 0',

},

wrapper: {

border: `${colors[type]} solid 1px`,

marginBottom: 15,

padding: 15,

position: 'relative',

}

}

return(

<div style={style.wrapper}>

<h2 style={style.heading}>{title}</h2>

{children}

</div>

)

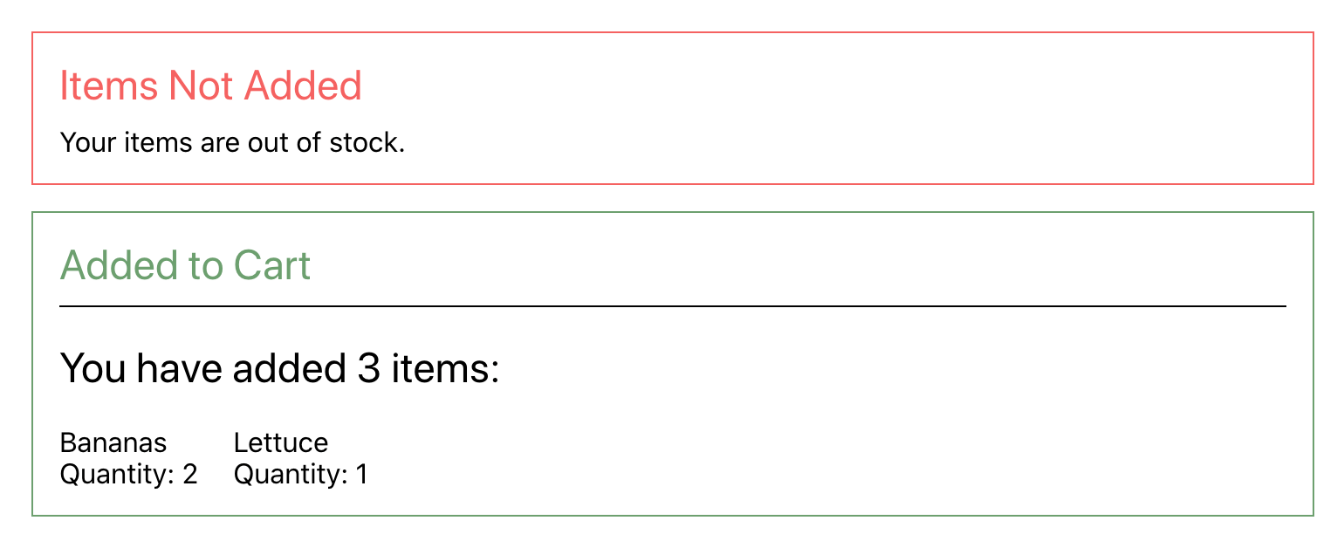
}

...

Copie

Il y a quelques changements ici. Tout d'abord, vous utilisez une déclaration de style unique pour wrapper, puis définissez dynamiquement la couleur en fonction du type. Vous n'êtes pas en train de styliser des <h2>éléments en général, mais plutôt de styliser ces éléments particuliers, qui se trouvent être des <h2>éléments. Étant donné que vous n'appliquez pas de styles à un type d'élément, il n'y a aucun risque que les styles descendent jusqu'aux éléments enfants.

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez les styles appliqués.



Les objets de style résolvent de nombreux problèmes, mais ils présentent des inconvénients. Premièrement, il [y a un coût de performance pour les styles en ligne](https://reactjs.org/docs/faq-styling.html#are-inline-styles-bad) . Les navigateurs ont été conçus pour gérer efficacement les CSS, et les objets de style qui appliquent des styles en ligne ne peuvent pas tirer parti de ces optimisations. L'autre problème est qu'il est plus difficile d'appliquer des styles aux éléments enfants avec des objets de style. Dans ce cas, vous ne vouliez pas qu'un style s'applique aux enfants, mais vous souhaitiez souvent que les styles soient mis en cascade. Par exemple, définir une famille de polices personnalisée sur chaque élément ou appliquer une taille personnalisée à chaque <h2>élément serait plus facile si vous utilisiez une stratégie de style moins spécifique.

Il existe cependant un juste milieu entre ces approches. Plusieurs bibliothèques tierces sont conçues pour trouver ce juste milieu. Dans l'étape suivante, vous allez créer des styles à l'aide d'une approche hybride appelée CSS-in-JS à l'aide d'une bibliothèque appelée JSS.

Étape 4 - Styling avec JSS

Dans cette étape, vous allez styliser des objets à l'aide de la bibliothèque populaire [JSS](https://cssinjs.org/) . Vous allez installer la nouvelle bibliothèque et convertir vos objets de style en objets JSS. Vous refactoriserez ensuite votre code pour utiliser des noms de classe générés dynamiquement qui éviteront les conflits entre les noms de classe à travers les modules. Vous allez également créer des objets de style JavaScript qui définissent dynamiquement des styles et utilisent des propriétés imbriquées pour créer des règles de style spécifiques.

JSS est une bibliothèque pour créer CSS-in-JS. Cette méthodologie a de nombreux [cas d'utilisation et options différents](https://cssinjs.org/from-sass-to-cssinjs) , mais le principal avantage de ce didacticiel est qu'elle crée des noms de classe dynamiques qui évitent les conflits entre les composants. Vous pourrez également profiter de la syntaxe JavaScript, ce qui signifie que vous pourrez utiliser des variables et créer des styles basés sur les accessoires React.

Pour commencer, installez la [version spécifique de React de JSS](https://cssinjs.org/react-jss/?v=v10.1.1) . Ce tutoriel utilisera la version 10.1.1:

* npm install react-jss

Copie

Le package installera plusieurs dépendances, y compris un certain nombre de [plugins JSS](https://cssinjs.org/plugins?v=v10.1.1) qui vous donneront la possibilité d'écrire des règles de style concises.

Une fois l'installation terminée, vous verrez un message de réussite:

Output

+ react-jss@10.1.1

added 281 packages from 178 contributors, removed 142 packages, updated 1392 packages and audited 1025251 packages in 144.872s

Votre sortie variera légèrement en fonction de la version de votre Node et d'autres dépendances.

Maintenant que la bibliothèque est installée, convertissez-la App.jspour utiliser JSS. Tout d'abord, ouvrez App.js:

* nano src/components/App/App.js

Copie

Il y a deux étapes pour utiliser JSS. Tout d'abord, vous devez importer une fonction pour créer un *[hook personnalisé](https://reactjs.org/docs/hooks-custom.html)* . Les hooks sont des fonctions que React exécutera sur chaque rendu de composant. Avec JSS, vous devez créer un hook en passant les définitions de style, en dehors du composant. Cela empêchera le code de s'exécuter à chaque nouveau rendu; puisque les définitions de style sont statiques, il n'y a aucune raison d'exécuter le code plus d'une fois.

Créez le crochet et l'objet de style en apportant les modifications en surbrillance:

styling-tutorial / src / components / App / App.js

import React from 'react';

import { createUseStyles } from 'react-jss';

import Alert from '../Alert/Alert';

import CartSuccess from '../CartSuccess/CartSuccess';

const useStyles = createUseStyles({

wrapper: {

padding: 20,

}

});

function App() {

return(

<div>

<Alert title="Items Not Added" type="error">

<div>Your items are out of stock.</div>

</Alert>

<CartSuccess />

</div>

)

}

export default App;

Copie

Notez dans ce cas que votre objet de style contient un autre objet appelé wrapper, qui contient les styles utilisant le même format de cas de chameau. Le nom de l'objet wrapper- - sert de base à la création du nom de classe dynamique.

Après avoir créé le hook, vous le consommez en exécutant la fonction à l'intérieur du composant. Cela enregistre le hook et crée les styles de manière dynamique. Effectuez la modification suivante en surbrillance:

styling-tutorial / src / components / App / App.js

import React from 'react';

import { createUseStyles } from 'react-jss'

import Alert from '../Alert/Alert';

import CartSuccess from '../CartSuccess/CartSuccess';

const useStyles = createUseStyles({

wrapper: {

padding: 20,

}

});

function App() {

const classes = useStyles()

return(

<div className={classes.wrapper}>

<Alert title="Items Not Added" type="error">

<div>Your items are out of stock.</div>

</Alert>

<CartSuccess />

</div>

)

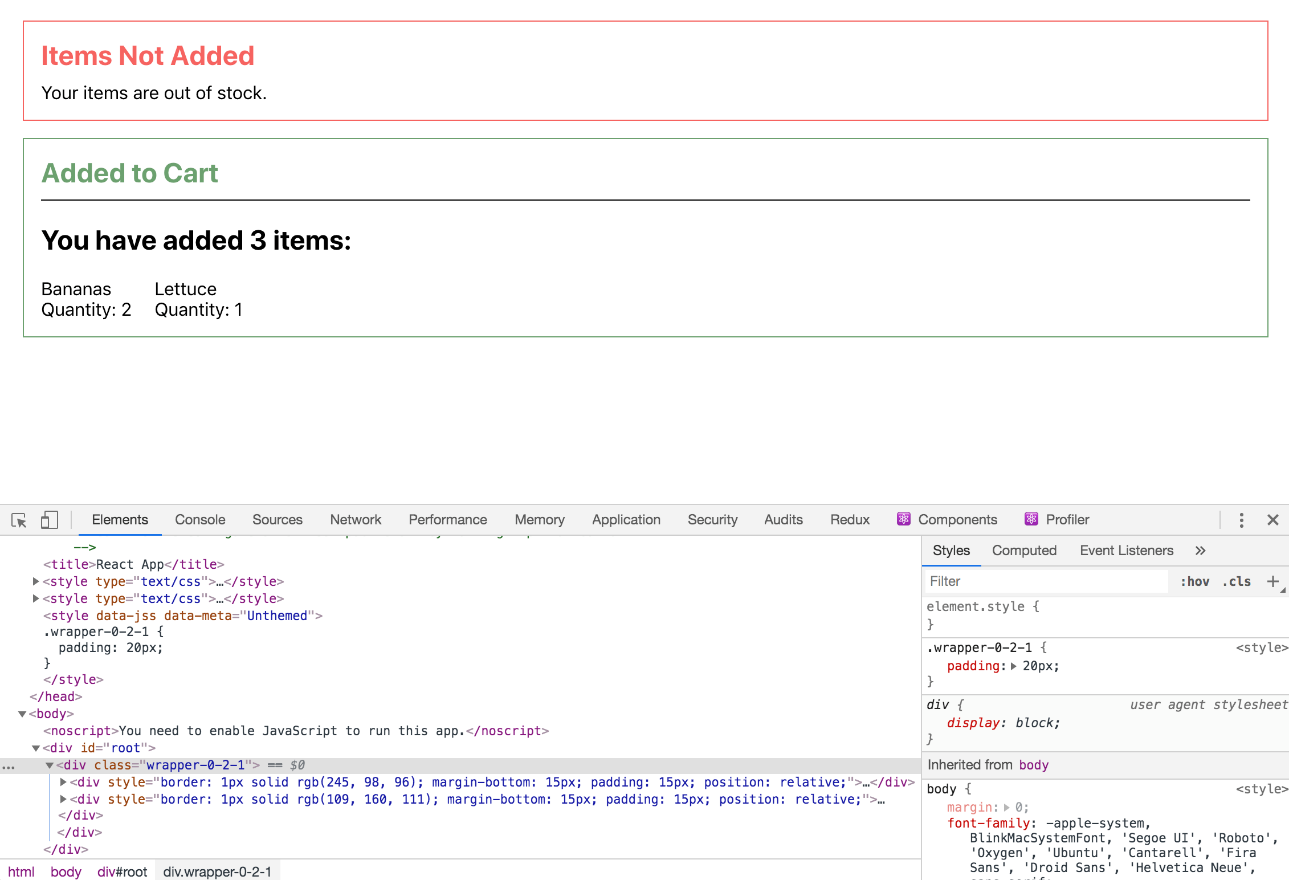
}

export default App;

Copie

Dans ce code, vous appelez la fonction et affectez les résultats à une variable appelée classes. La nouvelle variable classessera un objet contenant les noms de classe dynamiques. Vous appliquez ensuite le nom de classe approprié à votre élément en utilisant le même nom que celui que vous avez défini sur l'objet. Ici vous avez utilisé classes.wrapper.

Enregistrez et fermez le fichier. Lorsque vous le faites, le navigateur s'actualisera et vous verrez les mêmes styles qu'avant. Mais si vous regardez la console, vous verrez que le nom de la classe ne correspond pas parfaitement à celui que vous avez défini dans votre objet:



Dans ce cas, le nom de la classe est wrapper-0-2-1, mais votre nom de classe peut être différent. Vous verrez également que les styles sont convertis d'un objet en CSS et placés dans une <style>balise. Comparez cela aux styles en ligne, qui ne sont pas convertis en CSS et n'ont aucun nom de classe.

JSS crée les noms de classe de manière dynamique afin qu'ils n'entrent pas en conflit avec des noms similaires dans d'autres fichiers. Pour voir cela au travail, refactorisez CartSuccess.jspour utiliser le style JSS.

Ouvrez le fichier:

* nano src/components/CartSuccess/CartSuccess.js

Copie

À l'intérieur du fichier, créez un hook personnalisé à l'aide de createUseStyles. Au lieu d'appliquer une classe à l' <h2>élément, vous allez créer une règle pour les <h2>éléments à l'intérieur d'un wrapper. Pour ce faire avec du CSS simple, vous ajoutez un espace entre la classe et l'élément— .wrapper h2. Cela applique le style à tous les <h2>éléments qui sont des enfants de la .wrapperclasse.

Avec JSS, vous pouvez créer une règle similaire en créant un autre objet à l'intérieur de l'élément conteneur. Pour les lier, démarrez le nom de l'objet avec le &symbole:

styling-tutorial / src / components / CartSuccess / CartSuccess.js

import React from 'react';

import { createUseStyles } from 'react-jss';

import Alert from '../Alert/Alert';

const useStyles = createUseStyles({

item: {

marginRight: 20

},

wrapper: {

borderTop: 'black solid 1px',

display: 'flex',

flexWrap: 'wrap',

'& h2': {

width: '100%'

}

}

})

export default function CartSuccess() {

const classes = useStyles();

return(

<Alert title="Added to Cart" type="success">

<div className={classes.wrapper}>

<h2>

You have added 3 items:

</h2>

<div className={classes.item}>

<div>Bananas</div>

<div>Quantity: 2</div>

</div>

<div className={classes.item}>

<div>Lettuce</div>

<div>Quantity: 1</div>

</div>

</div>

</Alert>

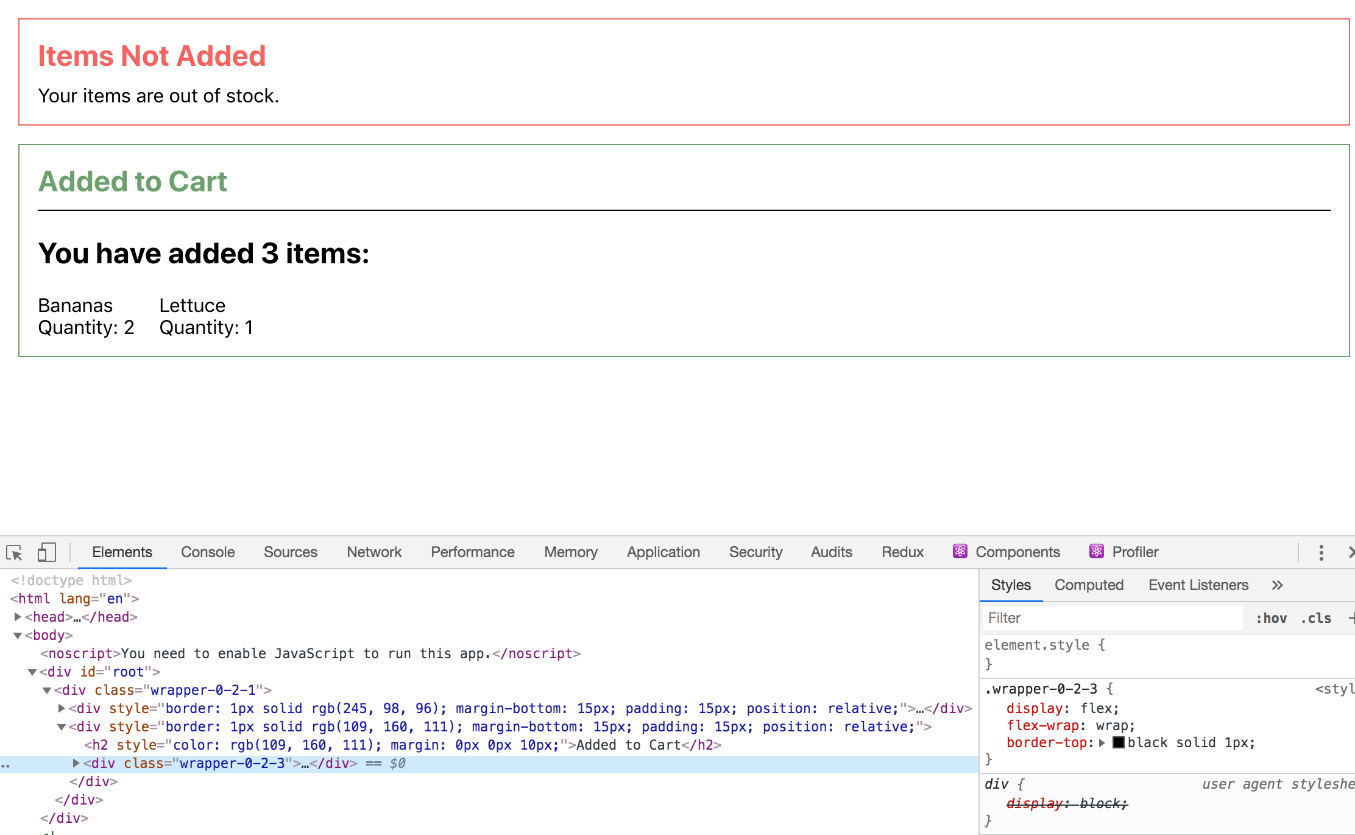
)

}

Copie

En plus de créer des règles pour le wrapper, vous avez également créé une règle pour item. Après avoir créé le hook personnalisé, vous avez transmis les noms de classe personnalisés à la classNamepropriété.

Enregistrez le fichier. Notez que vous utilisez le même nom — wrapper—dans ce composant et dans le Appcomposant. Mais lorsque le navigateur se rechargera, il n'y aura pas de conflit de dénomination; tout aura l'air correct. Si vous inspectez les éléments dans votre navigateur, vous verrez que même s'ils ont commencé avec le même nom, ils ont chacun une classe unique:



Dans ce cas, la classe du composant externe est wrapper-0-2-1, qui a été générée dans le Appcomposant. La classe pour CartSuccessest wrapper-0-2-3. Les noms de vos composants peuvent être légèrement différents, mais ils seront uniques.

Dans certaines situations, vous devrez peut-être créer un sélecteur spécifique pour remplacer d'autres styles. Par exemple, disons que vous souhaitez que le style item s'applique uniquement lorsque l'élément est un enfant de la classe wrapper. Pour ce faire, créez d'abord la classe sur l'objet sans propriétés. Ensuite, à l'intérieur de la classe wrapper, référencez la nouvelle classe avec un $symbole:

styling-tutorial / src / components / CartSuccess / CartSuccess.js

import React from 'react';

import { createUseStyles } from 'react-jss'

import Alert from '../Alert/Alert';

const useStyles = createUseStyles({

item: {},

wrapper: {

borderTop: 'black solid 1px',

display: 'flex',

flexWrap: 'wrap',

'& h2': {

width: '100%'

},

'& $item': {

marginRight: 20

}

}

})

export default function CartSuccess() {

const classes = useStyles()

return(

<Alert title="Added to Cart" type="success">

<div className={classes.wrapper}>

<h2>

You have added 3 items:

</h2>

<div className={classes.item}>

<div>Bananas</div>

<div>Quantity: 2</div>

</div>

<div className={classes.item}>

<div>Lettuce</div>

<div>Quantity: 1</div>

</div>

</div>

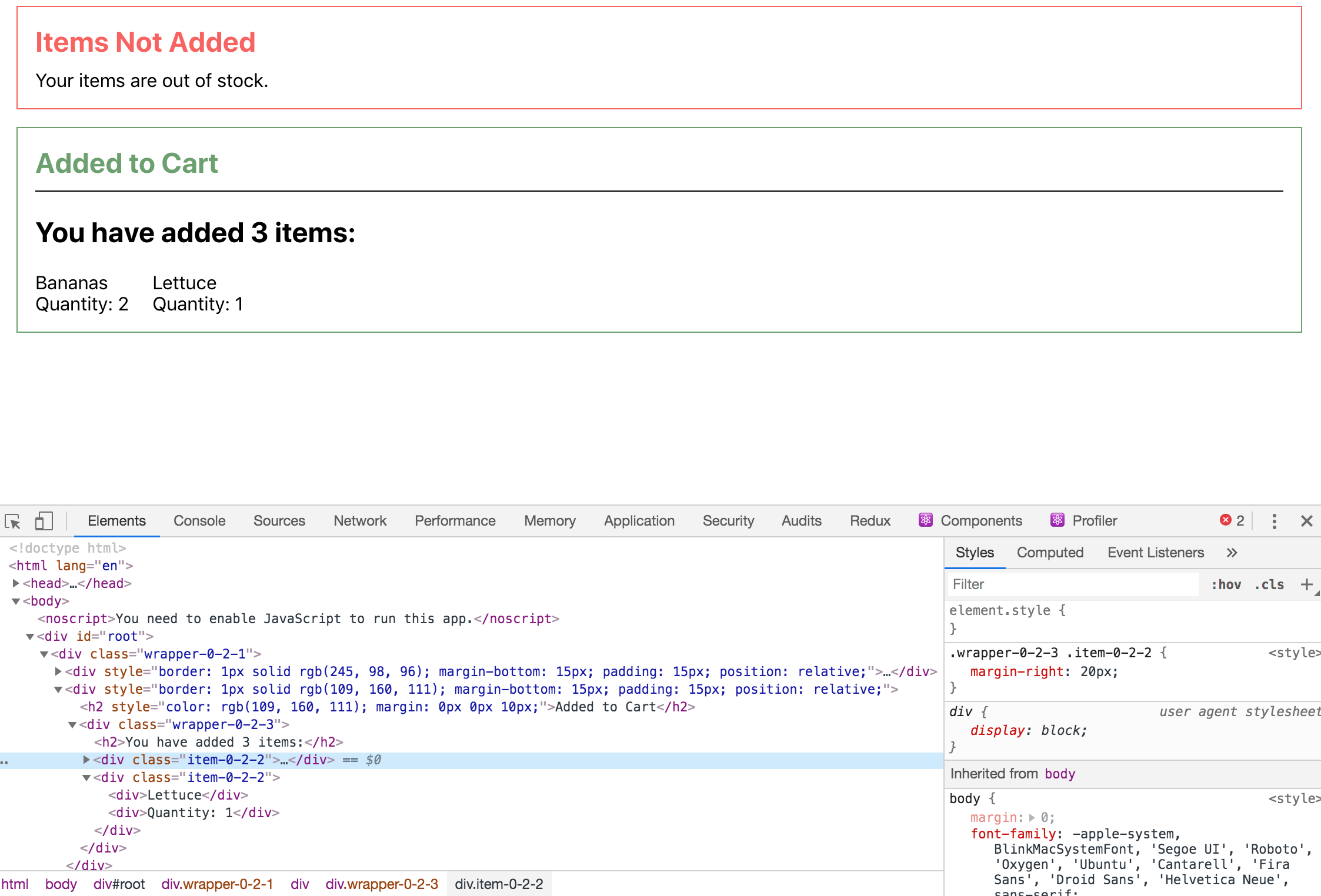
</Alert>

)

}

Copie

Enregistrez et fermez le fichier. Lorsque le navigateur se recharge, la page aura la même apparence, mais le itemCSS sera appliqué plus spécifiquement aux éléments sous le composant wrapper:



JSS vous donne la possibilité de créer des règles avec le même niveau de concentration que vous auriez créé avec du CSS ordinaire, mais le fera tout en créant des noms de classe uniques qui ne s'affronteront pas.

Un dernier avantage de JSS est que vous avez la possibilité d'utiliser des variables et d'autres fonctionnalités du langage JavaScript. Depuis que vous utilisez react-jss, vous pouvez transmettre des accessoires à l'objet de style pour créer des styles dynamiques. Pour tester cela, refactorisez le composant Alert.js pour utiliser des accessoires et des variables pour créer des propriétés dynamiques.

Tout d'abord, ouvrez le fichier:

* nano src/components/Alert/Alert.js

Copie

Créez un objet de style comme vous l'avez fait dans le dernier code refactoré. Assurez-vous de déplacer l'objet définissant les couleurs en dehors de la fonction du composant afin qu'il soit dans la même [portée](https://www.digitalocean.com/community/tutorials/understanding-variables-scope-hoisting-in-javascript#variable-scope) que la fonction createUseStyles:

styling-tutorial / src / components / Alert / Alert.js

import React from 'react';

import PropTypes from 'prop-types';

import { createUseStyles } from 'react-jss';

const colors = {

success: '#6da06f',

error: '#f56260',

};

const useStyles = createUseStyles({

wrapper: {

border: ({ type }) => `${colors[type]} solid 1px`,

marginBottom: 15,

padding: 15,

position: 'relative',

'& h2': {

color: ({ type }) => colors[type],

margin: [0, 0, 10, 0],

}

}

});

export default function Alert({ children, type, title }) {

const classes = useStyles({ type })

return(

<div className={classes.wrapper}>

<h2>{title}</h2>

{children}

</div>

)

}

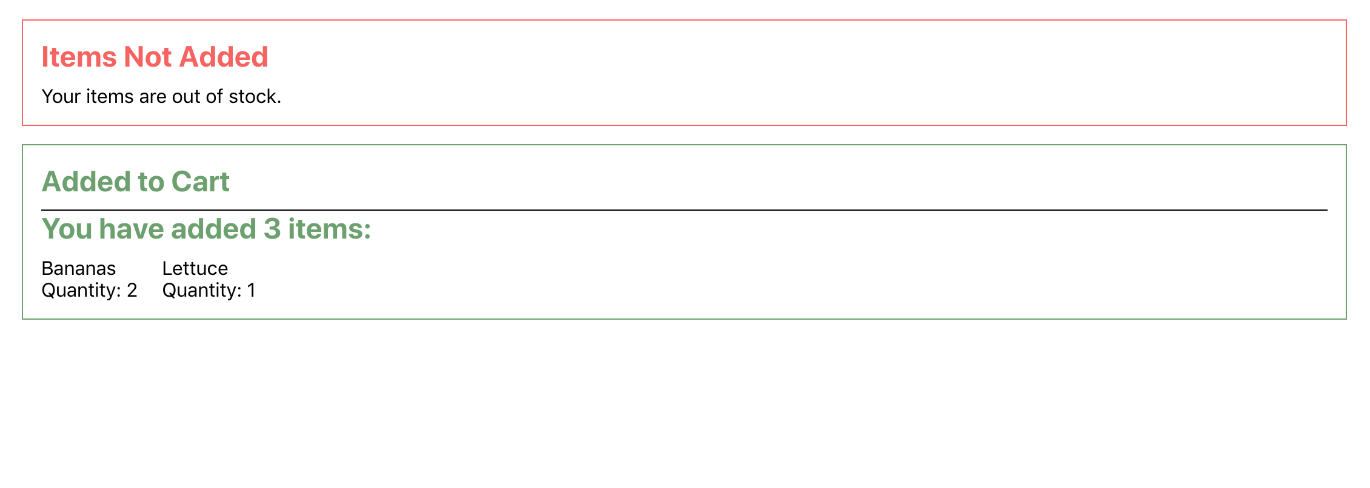
...

Copie

Pour passer des accessoires, vous faites de la règle de style une fonction. La fonction accepte les accessoires comme argument puis renvoie une règle. Pour créer une bordure dynamique, vous ajoutez bordercomme nom de propriété et une [fonction de flèche](https://www.digitalocean.com/community/tutorials/how-to-define-functions-in-javascript#arrow-functions) qui prend typeet retourne une chaîne: ({ type }) => `${colors[type]} solid 1px`,. Ensuite, après avoir créé votre hook, vous passez les accessoires que vous souhaitez référencer lors de la création de l'objet classes. Comme précédemment, vous stylisez la <h2>balise par élément au lieu de créer une classe spécifique. Vous transmettez également un tableau de valeurs pour marginplutôt qu'une chaîne telle que 0px 0px 10px 10px.

Enregistrez le fichier. Notez que vous n'avez pas à passer tous les accessoires dans la fonction. Dans ce cas, vous ne souhaitez utiliser que type, c'est donc tout ce dont vous avez besoin pour réussir. Cependant, vous pouvez passer plus ou même passer des accessoires inconnus en utilisant l'opérateur rest pour collecter des accessoires, puis les transmettre en groupe. Vous devez le passer en tant qu'objet; cependant, puisque c'est la manière standard de passer des accessoires, cela facilitera l'extension des arguments à l'avenir.

Lorsque la page se rechargera, vous verrez les couleurs correctes, mais il y aura un léger problème: la couleur verte de réussite met maintenant à jour l' <h2>élément dans CartSuccess:



JSS résout de nombreux problèmes, mais il crée toujours du CSS standard. Cela signifie que les styles peuvent s'appliquer aux éléments enfants si vous ne faites pas attention. Pour résoudre ce problème, ajoutez le >symbole pour que le CSS s'applique uniquement aux enfants immédiats:

styling-tutorial / src / components / Alert / Alert.js

import React from 'react';

...

const useStyles = createUseStyles({

wrapper: {

border: ({ type }) => `${colors[type]} solid 1px`,

marginBottom: 15,

padding: 15,

position: 'relative',

'& > h2': {

color: ({ type }) => colors[type],

margin: [0, 0, 10, 0],

}

}

});

export default function Alert({ children, type, title }) {

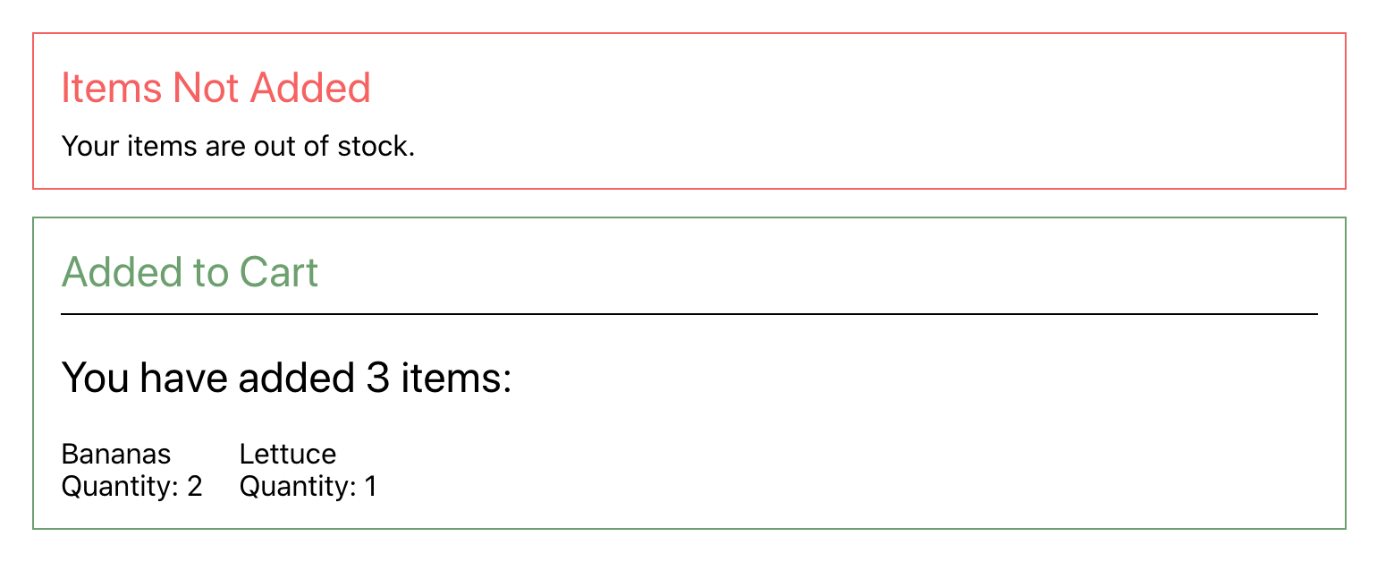
...

}

...

Copie

Enregistrez et fermez le fichier. Lorsque vous faites, le navigateur se rechargera et vous verrez les styles corrects:



Il y a beaucoup plus à JSS au-delà de ce qui est couvert dans ce didacticiel. Un avantage important que nous n'avons pas abordé est la [thématisation](https://cssinjs.org/react-jss/?v=v10.1.1#theming) . JSS vous permet de créer des styles basés sur des objets de thème prédéfinis. Cela signifie qu'au lieu de créer une couleur rouge à partir d'une valeur codée en dur, vous pouvez faire de la bordure d'alerte la alertcouleur, qui sera probablement une nuance de rouge, mais pourrait être différente en fonction de la définition du thème. Ceci est utile lors de la création de produits en marque blanche ou de la création de composants réutilisables qui doivent fonctionner dans tous les projets.

Dans cette étape, vous avez stylisé des composants à l'aide d'une bibliothèque tierce appelée react-jss. Vous avez également créé un objet de style et utilisé JSS pour convertir ces objets en classes dynamiques afin d'éviter tout conflit avec d'autres composants. En utilisant cette méthode, vous pouvez réutiliser en toute sécurité des noms de classes simples sans vous soucier des conflits plus tard dans le code. Enfin, vous avez appris à créer des styles à l'aide de fonctions et d'accessoires pour créer des styles dynamiques faisant référence aux accessoires de composant.

Conclusion

Tout au long de ce didacticiel, vous avez développé plusieurs composants réutilisables qui utilisent différentes techniques de style. Vous avez appris comment les objets de style et JSS créent des objets en utilisant des noms qui reflètent étroitement les propriétés CSS standard, et vous avez créé des composants qui peuvent définir dynamiquement des styles en fonction des propriétés entrantes. Vous avez également appris comment différentes approches offrent différentes options pour gérer les conflits de noms et la réutilisation.

Comme pour la plupart des techniques React, il n'y a pas de meilleure solution. Au lieu de cela, vous pouvez choisir l'option de style qui convient le mieux à votre projet. Avec ces options en main, vous pouvez commencer par quelque chose de simple et refactoriser au fur et à mesure que le projet se développe ou que les exigences changent, tout en restant confiant que vos composants continueront à répondre à vos objectifs de style.

Si vous souhaitez consulter d'autres didacticiels React, consultez notre [page React Topic](https://www.digitalocean.com/community/tags/react) ou revenez à la [page de la série Comment coder dans React.js](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-react-js) .