**Comment créer des composants Wrapper dans React avec des props**

Introduction

Dans ce tutoriel, vous allez créer des composants Wrapper avec des props en utilisant la [bibliothèque JavaScript React](https://reactjs.org/). Les *composants Wrapper* sont des composants qui entourent les composants inconnus et fournissent une structure par défaut pour afficher les composants enfants. Ce modèle est utile pour créer des éléments d'interface utilisateur (IU) qui sont utilisés de manière répétée tout au long d'un design, comme les modaux, les pages modèles et les tuiles d'information.

Pour créer des composants Wrapper, vous apprendrez d'abord à utiliser les [opérateurs rest et spread](https://www.digitalocean.com/community/tutorials/understanding-destructuring-rest-parameters-and-spread-syntax-in-javascript#spread) pour collecter les props non utilisés afin de les transmettre aux composants imbriqués. Ensuite, vous allez créer un composant qui utilise le composant children intégré pour envelopper les composants imbriqués dans [JSX](https://www.digitalocean.com/community/tutorials/how-to-create-react-elements-with-jsx) comme s'ils étaient des éléments [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML). Enfin, vous ferez passer des composants comme props pour créer des wrappers flexibles qui peuvent intégrer des JSX personnalisés à plusieurs endroits dans un composant.

Au cours du tutoriel, vous construirez des composants permettant d'afficher une liste de données sur les animaux sous forme de cartes. Vous apprendrez à diviser les données et à remodeler les composants tout en créant des composants d'emballage flexibles. À la fin de ce tutoriel, vous disposerez d'une application fonctionnelle qui utilisera des techniques props avancées pour créer des composants réutilisables qui évolueront et s'adapteront en fonction de la croissance et des changements de votre application.

**Note** : La première étape consiste à mettre en place un projet vierge sur lequel vous allez construire l'exercice du tutoriel. Si vous avez déjà un projet de travail et que vous souhaitez directement travailler avec des props, commencez par [**l'étape 2**](https://www.digitalocean.com/community/tutorials/how-to-create-wrapper-components-in-react-with-props#step-2-%E2%80%94-collecting-unused-props-with-props).

Conditions préalables

* Vous aurez besoin d'un environnement de développement qui exécute [Node.js](https://nodejs.org/en/about/) ; ce tutoriel a été testé sur Node.js version 10.19.0 et npm version 6.13.4. Pour l'installer sur macOS ou Ubuntu 18.04, suivez les étapes du tutoriel [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** du tutoriel [Comment installer Node.js sur Ubuntu 18.04](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-18-04#installing-using-a-ppa).
* Dans ce tutoriel, vous allez créer une application avec [Create React App](https://github.com/facebook/create-react-app). Vous trouverez les instructions pour installer une application avec Create React App et des informations générales sur son fonctionnement dans [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 les 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). Il vous aidera également à avoir une compréhension de base des props React, que vous pouvez apprendre dans [Comment Personnaliser les Composants React avec des Props](https://www.digitalocean.com/community/tutorials/how-to-customize-react-components-with-props).
* Vous devrez également avoir une connaissance de base de JavaScript, que vous pouvez trouver dans [Comment coder en JavaScript](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-javascript), ainsi qu'une connaissance de base du HTML et du CSS. Le [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML) est une bonne ressource pour le HTML et le CSS.

Étape 1 - Créer un projet vide

Dans cette étape, vous allez créer un nouveau projet en utilisant [Create React App](https://github.com/facebook/create-react-app). Ensuite, vous supprimerez le projet type et les fichiers connexes qui sont installés lorsque vous démarrez le projet. Enfin, vous créerez une structure de fichiers simple pour organiser vos éléments. Cela vous donnera une base solide sur laquelle vous pourrez construire l'application wrapper de ce tutoriel lors de la prochaine étape.

Pour commencer, faites un nouveau projet. Dans votre ligne de commande, exécutez le script suivant pour installer un nouveau projet en utilisant create-react-app :

* npx create-react-app wrapper-tutorial

Copy

Une fois le projet terminé, passez dans le répertoire :

* cd wrapper-tutorial

Copy

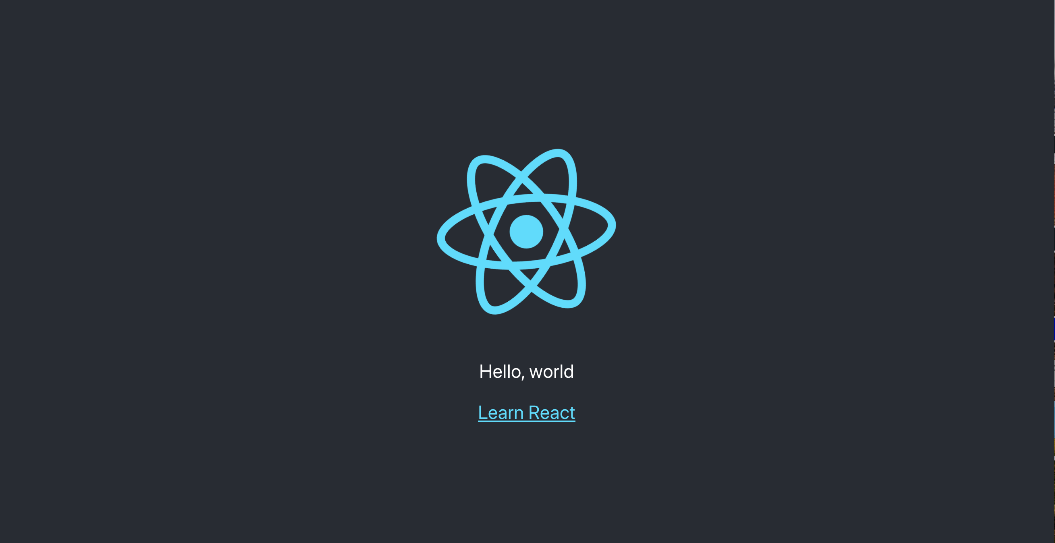
Dans un nouvel onglet ou une nouvelle fenêtre du terminal, démarrez le projet en utilisant le [script de démarrage de l'application Create React](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-create-react-app#step-3-%E2%80%94-starting-the-server) . Le navigateur se réactualise automatiquement en fonction des changements ; par conséquent, laissez ce script s'exécuter pendant que vous travaillez :

* npm start

Copy

Vous obtiendrez un serveur local en fonctionnement. Si le projet ne s'est pas ouvert dans une fenêtre de navigateur, vous pouvez l'ouvrir en naviguant vers <http://localhost:3000/>. Si vous l'exécutez à partir d'un serveur distant, l'adresse serahttp://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 passe-partout afin d'avoir un projet vide.

Pour commencer, ouvrez src/App.js dans un éditeur de texte. C'est le composant root (racine) qui est injecté dans la page. Tous les composants partiront de là. Vous pouvez trouver plus d'informations sur les App.js à [Comment mettre en place un projet React avec Create React App](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.js avec la commande suivante :

* nano src/App.js

Copy

Vous verrez un fichier comme ceci :

wrapper-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;

Copy

Supprimez le logo d'importation de ligne de '/logo.svg';   Remplacez ensuite tout ce qui figure dans la déclaration de retour pour renvoyer un ensemble de balises vides : <></>. Vous obtiendrez ainsi une page valide qui ne renvoie rien. Le code final ressemblera à ceci :

wrapper-tutorial/src/App.js

import React from 'react';

import './App.css';

function App() {

return <></>;

}

export default App;

Copy

Sauvegardez et quittez l'éditeur de texte.

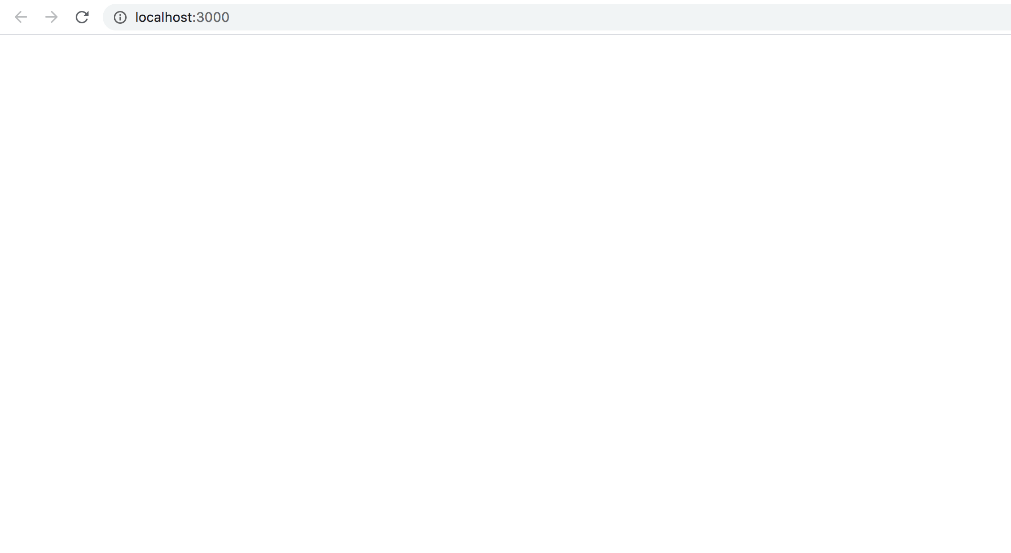
Enfin, supprimez le logo. Vous ne l'utiliserez pas dans votre demande et vous devez supprimer les fichiers inutilisés au fur et à mesure de votre travail. Cela vous évitera toute confusion sur le long terme.

Dans la fenêtre du terminal, tapez la commande suivante :

* rm src/logo.svg

Copy

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



Maintenant que vous avez éliminé l'exemple de projet Create React App, créez une structure de fichiers simple. Cela vous aidera à maintenir l'isolement et l'indépendance de vos composants.

Créez un répertoire appelé components dans le répertoire src. Celui-ci contiendra tous vos composants personnalisés.

* mkdir src/components

Copy

Chaque composant aura son propre répertoire pour stocker le fichier du composant ainsi que les styles, les images s'il y en a, et les tests.

Créez un répertoire pour App :

* mkdir src/components/App

Copy

Déplacez tous les fichiers App dans ce répertoire. Utilisez le joker, \*, pour sélectionner tous les fichiers qui commencent par App. quelle que soit leur extension. Utilisez ensuite la commande mv pour les placer dans le nouveau répertoire.

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

Copy

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

* nano src/index.js

Copy

La déclaration d'importation doit pointer vers le fichier App.js dans le répertoire App, donc faites la modification suivante en surbrillance :

wrapper-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();

Copy

Enregistrez et quittez le fichier.

Maintenant que le projet est mis en place, vous pouvez créer votre premier composant.

Étape 2 — Collecte des Props non utilisés avec ...props

Dans cette étape, vous allez créer un composant pour afficher un ensemble de données sur un groupe d'animaux. Votre composant contiendra un deuxième composant imbriqué pour afficher certaines informations visuellement. Pour connecter le composant parent et le composant imbriqué, vous utiliserez les [opérateurs rest et spread](https://www.digitalocean.com/community/tutorials/understanding-destructuring-rest-parameters-and-spread-syntax-in-javascript#spread) pour faire passer les props inutilisés du parent à l'enfant sans que le parent ait besoin de connaître les noms ou les types des props.

À la fin de cette étape, vous aurez un composant parent qui sera en mesure de fournir des props aux composants imbriqués sans avoir à connaître ce que sont les props. Ainsi, le composant parent restera flexible, ce qui vous permettra de mettre à jour le composant enfant sans avoir à changer le parent.

Création d'un composant AnimalCard

Pour commencer, créez un ensemble de données pour vos animaux. Tout d'abord, ouvrez un fichier contenant l'ensemble des données dans le répertoire components/App :

* nano src/components/App/data.js

Copy

Ajoutez les données suivantes :

src/components/App/data.js

export default [

{

name: 'Lion',

scientificName: 'Panthero leo',

size: 140,

diet: ['meat']

},

{

name: 'Gorilla',

scientificName: 'Gorilla beringei',

size: 205,

diet: ['plants', 'insects']

},

{

name: 'Zebra',

scientificName: 'Equus quagga',

size: 322,

diet: ['plants'],

}

]

Copy

Cette liste d'animaux est un [array](https://www.digitalocean.com/community/tutorials/understanding-arrays-in-javascript) d’[objets](https://www.digitalocean.com/community/tutorials/understanding-objects-in-javascript) qui comprend le nom de l'animal, son nom scientifique, son poids et son régime alimentaire.

Enregistrez et fermez le fichier.

Ensuite, créez un répertoire pour le composant AnimalCard :

* mkdir src/components/AnimalCard

Copy

Ouvrez un nouveau dossier dans le répertoire :

* nano src/components/AnimalCard/AnimalCard.js

Copy

Ajoutez maintenant un composant qui prendra le nom, le régime alimentaire et la taille comme prop et l'affichera :

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

import React from 'react';

import PropTypes from 'prop-types';

export default function AnimalCard({ diet, name, size }) {

return(

<div>

<h3>{name}</h3>

<div>{size}kg</div>

<div>{diet.join(', ')}.</div>

</div>

)

}

AnimalCard.propTypes = {

diet: PropTypes.arrayOf(PropTypes.string).isRequired,

name: PropTypes.string.isRequired,

size: PropTypes.number.isRequired,

}

Copy

Ici, vous [déstructurez](https://www.digitalocean.com/community/tutorials/understanding-destructuring-rest-parameters-and-spread-syntax-in-javascript) les props dans la liste des paramètres de la fonction AnimalCard, puis vous affichez les données dans un div. Les données relatives au régime alimentaire sont répertoriées sous la forme d'une seule chaîne de caractères à l'aide de [la méthode join()](https://www.digitalocean.com/community/tutorials/how-to-use-array-methods-in-javascript-accessor-methods#join()). Chaque donnée comprend un [PropType](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) correspondant pour s'assurer que le type de données est correct.

Enregistrez et fermez le fichier.

Maintenant que vous avez votre composant et vos données, vous devez les combiner. Pour ce faire, importez le composant et les données dans le composant root de votre projet : App.js.

Tout d'abord, ouvrez le composant :

* nano src/components/App/App.js

Copy

De là, vous pouvez passer les données en boucle et renvoyer une nouvelle AnimalCard avec les props correspondantes. Ajoutez les lignes surlignées à App.js :

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

import React from 'react';

import './App.css';

import animals from './data';

import AnimalCard from '../AnimalCard/AnimalCard';

function App() {

return (

<div className="wrapper">

{animals.map(animal =>

<AnimalCard

diet={animal.diet}

key={animal.name}

name={animal.name}

size={animal.size}

/>

)}

</div>

);

}

export default App;

Copy

Enregistrez et fermez le fichier.

Au fur et à mesure que vous travaillerez sur des projets plus complexes, vos données proviendront d'endroits plus variés, tels que des [API](https://developer.mozilla.org/en-US/docs/Glossary/API), [des LocalStorage](https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Local_storage) ou des fichiers statiques. Mais le processus d'utilisation de chacun d'entre eux sera similaire : affecter les données à une variable et boucler les données. Dans ce cas, les données proviennent d'un fichier statique, donc vous les importez directement dans une variable.

Dans ce code, vous utilisez la méthode [.map()](https://www.digitalocean.com/community/tutorials/how-to-use-array-methods-in-javascript-iteration-methods#map()) pour itérer sur les animaux et afficher les props. Notez que vous n'êtes pas obligé d'utiliser toutes les données. Vous ne transmettez pas explicitement la propriété scientificName, par exemple. Vous ajoutez également une prop key séparée que React utilisera pour garder une [trace des données cartographiées](https://www.digitalocean.com/community/tutorials/how-to-create-react-elements-with-jsx#step-4-%E2%80%94-mapping-over-data-to-create-elements). Enfin, vous enveloppez le code avec un div avec un className de wrapper que vous utiliserez pour ajouter un peu de style.

Pour ajouter ce style, ouvrez App.css :

* nano src/components/App/App.css

Copy

Supprimez le style boilerplate et ajoutez [des propriétés flex](https://developer.mozilla.org/en-US/docs/Glossary/Flexbox) à une classe appelée wrapper :

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

.wrapper {

display: flex;

flex-wrap: wrap;

justify-content: space-between;

padding: 20px;

}

Copy

Les données seront organisées de manière à s'aligner grâce à une disposition en flexbox. padding donne de l'espace dans la fenêtre du navigateur et justify-content répartit l'espace supplémentaire entre les éléments.

Enregistrez et quittez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et vous verrez certaines données espacées.



Créer un composant Details

Vous disposez maintenant d'un simple composant qui affiche les données. Mais disons que vous vouliez donner un peu de style aux données sur l'alimentation en convertissant le texte en emoji. Vous pouvez le faire en convertissant les données dans votre composant.

React est conçu pour être flexible, de sorte que lorsque vous réfléchissez à la manière de convertir des données, vous disposez de plusieurs options différentes :

* Vous pouvez créer une fonction à l'intérieur du composant qui convertit le texte en un émoji.
* Vous pouvez créer une fonction et la stocker dans un fichier à l'extérieur du composant afin de pouvoir réutiliser la logique entre différents composants.
* Vous pouvez créer un composant distinct qui convertit le texte en un émoji.

Chaque approche est bonne lorsqu'elle est appliquée au bon cas d'utilisation, et vous vous retrouverez à passer de l'une à l'autre au fur et à mesure que vous construirez une application. Pour éviter une abstraction et une complexité prématurées, vous devez utiliser la première option pour commencer. Si vous souhaitez réutiliser la logique, vous pouvez retirer la fonction séparément du composant. La troisième option est la meilleure si vous voulez avoir une pièce réutilisable qui inclut la logique et le balisage, ou que vous voulez isoler pour l'utiliser dans toute l'application.

Dans ce cas-là, nous allons créer un nouveau composant, car nous voudrons ajouter d'autres données plus tard et nous combinons le balisage avec la logique de conversion.

Le nouveau composant s'appellera AnimalDetails. Pour ce faire, créez un nouveau répertoire :

* mkdir src/components/AnimalDetails

Copy

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

* nano src/components/AnimalDetails/AnimalDetails.js

Copy

À l'intérieur du fichier, créez un petit composant qui affiche la diet sous forme d'emoji :

wrapper-tutorial/src/components/AnimalDetails/AnimalDetails.js

import React from 'react';

import PropTypes from 'prop-types';

import './AnimalDetails.css';

function convertFood(food) {

switch(food) {

case 'insects':

return '🐜';

case 'meat':

return '🍖';

case 'plants':

default:

return '🌱';

}

}

export default function AnimalDetails({ diet }) {

return(

<div className="details">

<h4>Details:</h4>

<div>

Diet: {diet.map(food => convertFood(food)).join(' ')}

</div>

</div>

)

}

AnimalDetails.propTypes = {

diet: PropTypes.arrayOf(PropTypes.string).isRequired,

}

Copy

L'objet AnimalDetails.propTypes configure la fonction afin qu'elle prenne une prop de diet qui est un ensemble de chaines de caractères. Ensuite, à l'intérieur du composant, le code passe en boucle sur la diet et convertit la chaîne en émoji en utilisant l'instruction [switch](https://www.digitalocean.com/community/tutorials/how-to-use-the-switch-statement-in-javascript).

Enregistrez et fermez le fichier.

Vous importez également du CSS, alors ajoutons cela maintenant.

Ouvrez AnimalDetails.css :

* nano src/components/AnimalDetails/AnimalDetails.css

Copy

Ajoutez du CSS pour donner à l'élément une bordure et une marge pour séparer les détails du reste de l'élément :

wrapper-tutorial/src/components/AnimalDetails/AnimalDetails.css

.details {

border-top: gray solid 1px;

margin: 20px 0;

}

Copy

Nous utilisons .details pour faire correspondre la règle aux éléments avec une className de details.

Enregistrez et fermez le fichier.

Maintenant que vous avez un nouveau composant personnalisé, vous pouvez l'ajouter à votre composant AnimalCard. Ouvrez AnimalCard.js :

* nano src/components/AnimalCard/AnimalCard.js

Copy

Remplacez l'instruction diet.join par le nouveau composant AnimalDetails et passez diet comme une prop en ajoutant les lignes surlignées :

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

import React from 'react';

import PropTypes from 'prop-types';

import AnimalDetails from '../AnimalDetails/AnimalDetails';

export default function AnimalCard({ diet, name, size }) {

return(

<div>

<h3>{name}</h3>

<div>{size}kg</div>

<AnimalDetails

diet={diet}

/>

</div>

)

}

AnimalCard.propTypes = {

diet: PropTypes.arrayOf(PropTypes.string).isRequired,

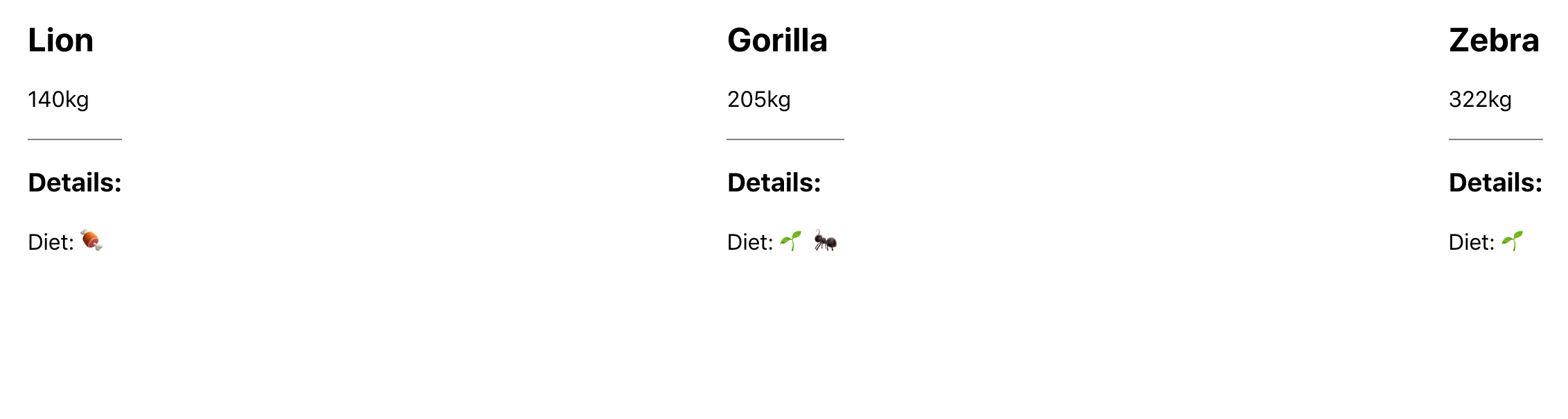
name: PropTypes.string.isRequired,

size: PropTypes.number.isRequired,

}

Copy

Enregistrez le fichier et vous verrez les nouveaux détails dans le navigateur.



Passage des détails à travers un composant avec ...props

Les composants fonctionnent bien ensemble, mais il y a une légère inefficacité dans AnimalCard. Vous retirez explicitement la diet de l'argument props, mais vous n'utilisez pas les données. Au lieu de cela, vous les transmettez au composant. Il n'y a rien de mal à cela en soi - en fait, il est souvent préférable de se tromper du côté d'une trop grande communication. Mais en faisant cela, vous rendez votre code plus difficile à maintenir. Chaque fois que vous voulez transmettre de nouvelles données à AnimalDetails, vous devez mettre à jour trois endroits : App, où vous passez les props, AnimalDetails, qui consomme les props, et AnimalCard, qui est l'intermédiaire.

Une meilleure façon de faire consiste à rassembler les props non utilisées dans AnimalCard et de les transmettre directement à AnimalDetails. Cela vous donne la possibilité d'apporter des modifications à AnimalDetails sans changer AnimalCard. En effet, AnimalCard n'a pas besoin de savoir quoi que ce soit sur les props ou les PropTypes qui vont dans AnimalDetails.

Pour ce faire, vous utiliserez l’[opérateur rest d'objet](https://www.digitalocean.com/community/tutorials/understanding-destructuring-rest-parameters-and-spread-syntax-in-javascript#rest-parameters). Cet opérateur collecte tous les objets qui ne sont pas retirés pendant la déstructuration et les enregistre dans un nouvel objet.

Voici un exemple simple :

const dog = {

name: 'dog',

diet: ['meat']

}

const { name, ...props } = dog;

Copy

Dans ce cas, le nom de la variable sera 'dog' et la variable props sera { diet: ['meat']}.

Jusqu'à présent, vous avez passé toutes les props comme s'ils étaient des attributs HTML, mais vous pouvez également utiliser des objets pour envoyer des props. Pour utiliser un objet en tant que prop, vous devez utiliser l'opérateur spread —...props— entourés de bretelles bouclées. Cela transformera chaque paire clé-valeur en une prop.

Ouvrez AnimalCard.js :

* nano src/components/AnimalCard/AnimalCard.js

Copy

À l'intérieur, retirez diet de l'objet déstructuré et rassemblez plutôt le reste des props dans une variable appelée props. Transmettez ensuite directement ces props à AnimalDetails :

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

import React from 'react';

import PropTypes from 'prop-types';

import AnimalDetails from '../AnimalDetails/AnimalDetails';

export default function AnimalCard({ name, size, ...props }) {

return(

<div>

<h3>{name}</h3>

<div>{size}kg</div>

<AnimalDetails

{...props}

/>

</div>

)

}

AnimalCard.propTypes = {

name: PropTypes.string.isRequired,

size: PropTypes.number.isRequired,

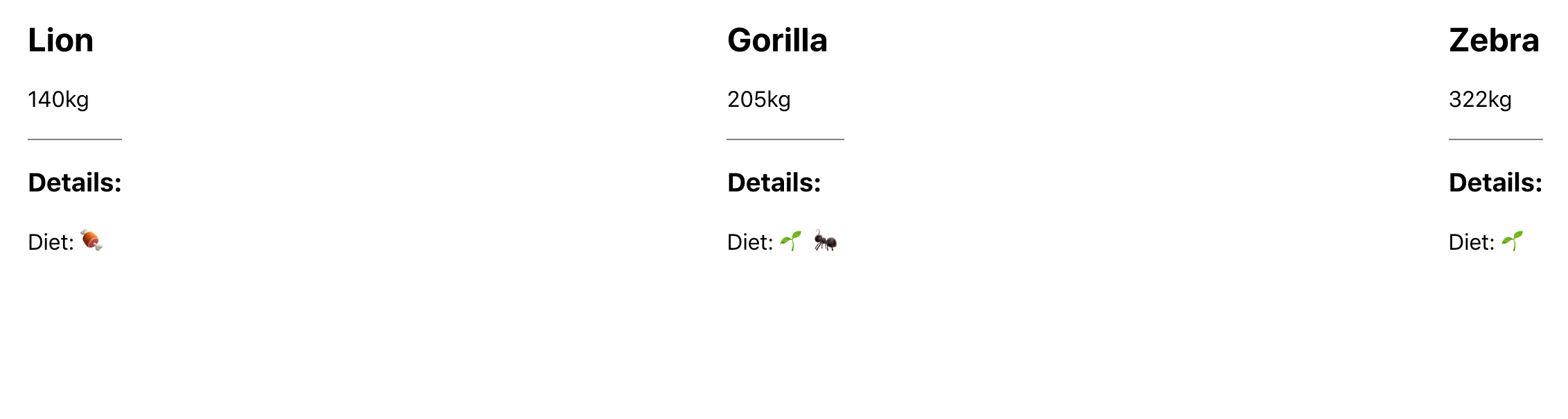
}

Copy

Notez que vous pouvez supprimer le PropType diet puisque vous n'utilisez pas la prop dans ce composant.

Dans ce cas, vous ne faites passer qu'une seule prop à AnimalDetails. Dans les cas où vous avez plusieurs props, l'ordre est important. Une prop ultérieure écrasera les props précédentes, donc si vous avez une prop que vous voulez privilégier, assurez-vous qu'il est le dernier. Cela peut entraîner une certaine confusion si votre objet props possède une propriété qui a également une valeur nominale.

Enregistrez et fermez le fichier. Le navigateur se rafraîchira et tout semblera similaire :



Pour voir comment l'objet ...props ajoute de la flexibilité, passons le scientificName à AnimalDetails via le composant AnimalCard.

Tout d'abord, ouvrez App.js :

* nano src/components/App/App.js

Copy

Passez ensuite le scientificName en tant que prop :

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

import React from 'react';

import './App.css';

import animals from './data';

import AnimalCard from '../AnimalCard/AnimalCard';

function App() {

return (

<div className="wrapper">

{animals.map(animal =>

<AnimalCard

diet={animal.diet}

key={animal.name}

name={animal.name}

size={animal.size}

scientificName={animal.scientificName}

/>

)}

</div>

);

}

export default App;

Copy

Enregistrez et fermez le fichier.

Passez sur AnimalCard ; vous n'aurez pas besoin d'y apporter de modifications. Ensuite, ouvrez AnimalDetails pour pouvoir consommer la nouvelle prop :

* nano src/components/AnimalDetails/AnimalDetails.js

Copy

La nouvelle prop sera une chaine de caractères, que vous ajouterez à la liste details avec une ligne déclarant le PropType :

wrapper-tutorial/src/components/AnimalDetails/AnimalDetails.js

import React from 'react';

...

export default function AnimalDetails({ diet, scientificName }) {

return(

<div className="details">

<h4>Details:</h4>

<div>

Scientific Name: {scientificName}.

</div>

<div>

Diet: {diet.map(food => convertFood(food)).join(' ')}

</div>

</div>

)

}

AnimalDetails.propTypes = {

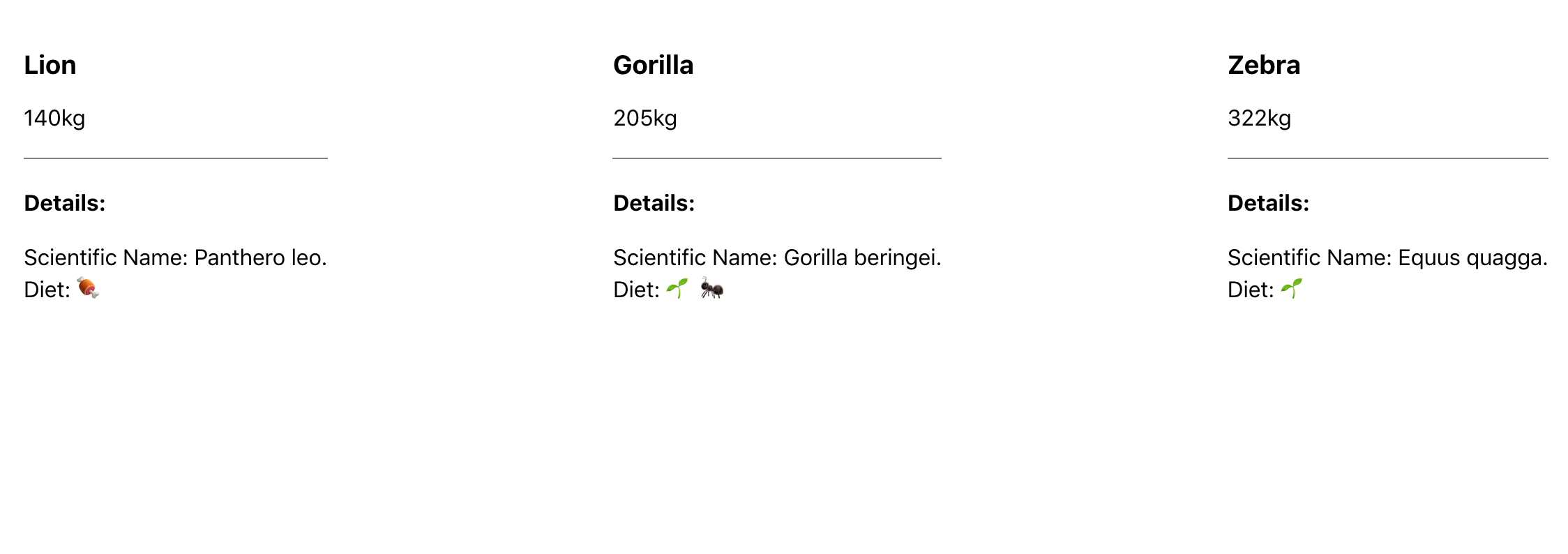
diet: PropTypes.arrayOf(PropTypes.string).isRequired,

scientificName: PropTypes.string.isRequired,

}

Copy

Enregistrez et fermez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et vous verrez les nouveaux détails sans aucune modification du composant AnimalCard :



Dans cette étape, vous avez appris à créer des props parents flexibles qui peuvent prendre des props inconnues et les faire passer dans des composants imbriqués avec l'opérateur spread. Il s'agit d'un modèle commun qui vous donnera la flexibilité nécessaire pour créer des composantes avec des responsabilités ciblées. Dans l'étape suivante, vous allez créer des composants qui peuvent prendre des composants inconnus comme prop en utilisant la prop children intégrée.

Étape 3 — Création de Composants Wrapper avec children

Dans cette étape, vous allez créer un composant wrapper qui peut prendre un groupe inconnu de composants comme prop. Cela vous donnera la possibilité d'imbriquer des composants comme du HTML standard, et vous donnera un modèle pour créer des wrappers réutilisables qui vous permettront de fabriquer une variété de composants qui ont besoin d'un design commun mais d'un intérieur flexible.

React vous donne une prop intégrée appelé children qui rassemble tous les composants enfants. L'utilisation de cette méthode rend la création de composants wrapper intuitive et lisible.

Pour commencer, fabriquez un nouveau composant appelé Card. Il s'agira d'un composant wrapper pour créer un style standard pour tout nouveau composant de carte.

Créez un nouveau répertoire :

* mkdir src/components/Card

Copy

Ensuite, ouvrez le composant Card dans votre éditeur de texte :

* nano src/components/Card/Card.js

Copy

Créez un composant qui prend children et title comme props et les enveloppe dans un div en ajoutant le code suivant :

wrapper-tutorial/src/components/Card/Card.js

import React from 'react';

import PropTypes from 'prop-types';

import './Card.css';

export default function Card({ children, title }) {

return(

<div className="card">

<div className="card-details">

<h2>{title}</h2>

</div>

{children}

</div>

)

}

Card.propTypes = {

children: PropTypes.oneOfType([

PropTypes.arrayOf(PropTypes.element),

PropTypes.element.isRequired

]),

title: PropTypes.string.isRequired,

}

Copy

Les PropTypes pour les children sont nouveaux. La prop children peut être soit un élément JSX, soit un ensemble d'éléments JSX. Le titre est une chaîne de caractères.

Enregistrez et fermez le fichier.

Ensuite, ajoutez un peu de style. Ouvrez Card.css :

* nano src/components/Card/Card.css

Copy

Votre carte aura une bordure et une ligne sous les détails.

wrapper-tutorial/src/components/Card/Card.css

.card {

border: black solid 1px;

margin: 10px;

padding: 10px;

width: 200px;

}

.card-details {

border-bottom: gray solid 1px;

margin-bottom: 20px;

}

Copy

Enregistrez et fermez le fichier. Maintenant que vous avez votre composant, vous devez l'utiliser. Vous pourriez emballer chaque AnimalCard avec le composant Card dans des App.js, mais puisque le nom AnimalCard implique qu'il s'agit déjà d'une Card, il serait préférable d'utiliser le composant Card à l'intérieur d’AnimalCard.

Ouvrez AnimalCard :

* nano src/components/AnimalCard/AnimalCard.js

Copy

Contrairement à d'autres props, on ne passe pas children explicitement. À la place, vous incluez les JSX comme s'ils étaient des éléments HTML enfants. En d'autres termes, il suffit de les emboîter dans l'élément, comme suit :

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

import React from 'react';

import PropTypes from 'prop-types';

import Card from '../Card/Card';

import AnimalDetails from '../AnimalDetails/AnimalDetails';

export default function AnimalCard({ name, size, ...props }) {

return(

<Card title="Animal">

<h3>{name}</h3>

<div>{size}kg</div>

<AnimalDetails

{...props}

/>

</Card>

)

}

AnimalCard.propTypes = {

name: PropTypes.string.isRequired,

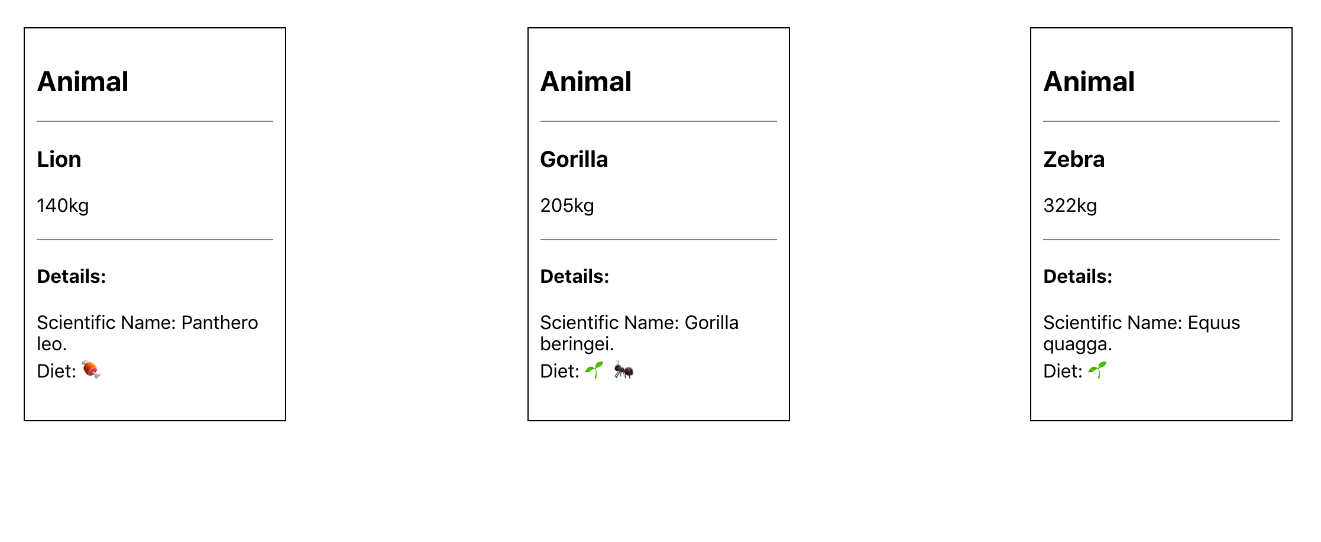
size: PropTypes.number.isRequired,

}

Copy

Contrairement à un composant React, vous n'avez pas besoin d'avoir un élément root unique comme enfant. C'est pourquoi le PropType pour Card a précisé qu'il pouvait s'agir d'un ensemble d'éléments ou d'un seul élément. En plus de faire passer les children comme des éléments emboîtés, vous donnez à la carte un titre d’Animal.

Enregistrez et fermez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et vous verrez le composant de carte mis à jour.



Vous disposez maintenant d'un composant Card réutilisable qui peut accueillir un nombre illimité d'enfants imbriqués. Le principal avantage est que vous pouvez réutiliser le composant Card avec n'importe quel composant arbitraire. Si vous vouliez faire une carte Plant, vous pourriez le faire en enveloppant les informations sur les plantes avec le composant Card. Il n'est même pas nécessaire qu'il existe un quelconque lien : si vous vouliez réutiliser le composant Card dans une application complètement différente qui répertorie des éléments tels que la musique ou les données de compte, vous pourriez le faire aussi. Le composant Card ne se soucie pas de ce que sont les enfants ; vous réutilisez simplement l'élément Emballage, qui dans ce cas est la bordure stylisée et le titre.

L'inconvénient de l'utilisation de children est que vous ne pouvez avoir qu'un seul exemplaire de la prop enfant. De temps en temps, vous voudrez qu'un composant ait un JSX personnalisé à plusieurs endroits. Heureusement, vous pouvez le faire en faisant passer des composants JSX et React comme props, technique que nous couvrirons dans la prochaine étape.

Étape 4 — Passation de Composants comme Props

Dans cette étape, vous modifierez le composant Card afin de prendre d'autres composants comme props. Cela donnera à votre composant un maximum de flexibilité pour afficher des composants inconnus ou JSX en plusieurs endroits de la page. Contrairement à children, que vous ne pouvez utiliser qu'une seule fois, vous pouvez avoir autant de composants que de props, ce qui donne à votre composant wrapper la capacité de s'adapter à une variété de besoins tout en conservant une apparence et une structure standard.

À la fin de cette étape, vous aurez un composant qui peut envelopper les composants enfants et aussi afficher d'autres composants dans la carte. Ce modèle vous donnera une certaine souplesse lorsque vous devrez créer des composants nécessitant des informations plus complexes que de simples chaînes de caractères et des entiers.

Modifions le composant Card pour prendre un élément React arbitraire appelé details.

Tout d'abord, ouvrez le composant Card :

* nano src/components/Card/Card.js

Copy

Ensuite, ajoutez un nouvel accessoire appelé details et placez-le sous l'élément <h2> :

wrapper-tutorial/src/components/Card/Card.js

import React from 'react';

import PropTypes from 'prop-types';

import './Card.css';

export default function Card({ children, details, title }) {

return(

<div className="card">

<div className="card-details">

<h2>{title}</h2>

{details}

</div>

{children}

</div>

)

}

Card.propTypes = {

children: PropTypes.oneOfType([

PropTypes.arrayOf(PropTypes.element),

PropTypes.element.isRequired

]),

details: PropTypes.element,

title: PropTypes.string.isRequired,

}

Card.defaultProps = {

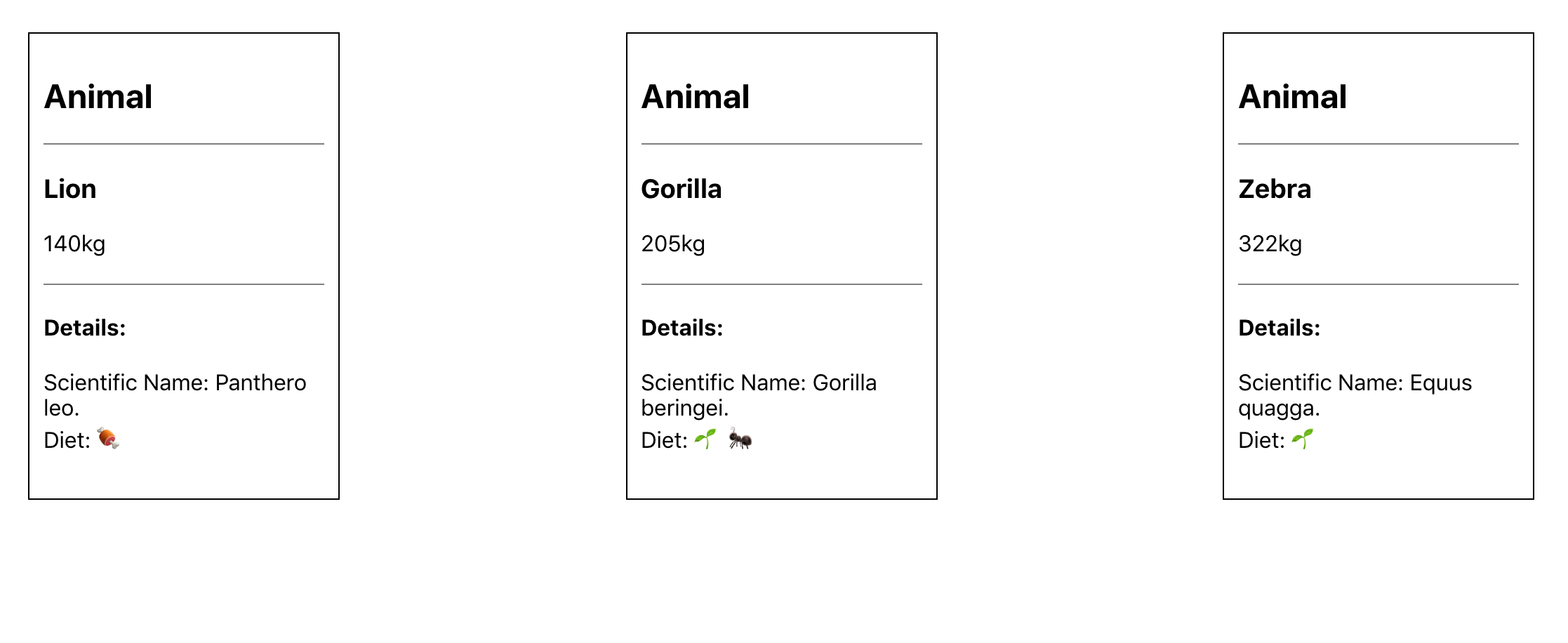
details: null,

}

Copy

Cette prop aura le même type que children, mais il devra être facultatif. Pour le rendre facultatif, vous ajoutez une valeur par défaut de null. Dans ce cas, si un utilisateur ne transmet aucun détail, le composant sera toujours valable et n'affichera rien de plus.

Enregistrez et fermez le fichier. La page se rafraîchira et vous verrez la même image qu'auparavant :



Ajoutez maintenant quelques détails à la AnimalCard. Tout d'abord, ouvrez AnimalCard.

* nano src/components/AnimalCard/AnimalCard.js

Copy

Comme le composant Card utilise déjà children, vous devrez passer le nouveau composant JSX comme prop. Comme ce sont tous des mammifères, ajoutez cela à la carte, mais enveloppez-la dans des étiquettes <em> pour la mettre en italique.

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

import React from 'react';

...

export default function AnimalCard({ name, size, ...props }) {

return(

<Card title="Animal" details={<em>Mammal</em>}>

<h3>{name}</h3>

<div>{size}kg</div>

<AnimalDetails

{...props}

/>

</Card>

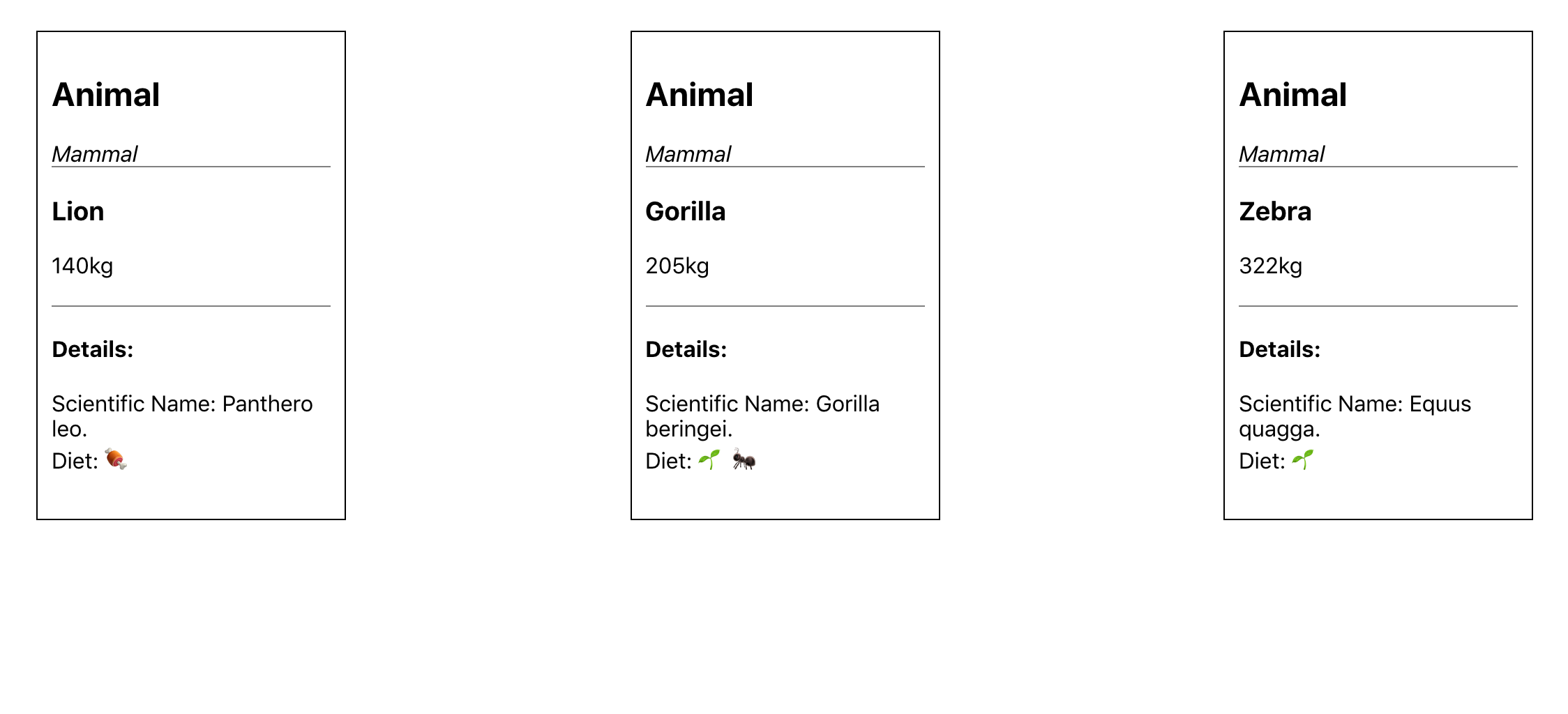
)

}

...

Copy

Sauvegardez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et vous verrez la mise à jour, y compris l'expression **Mammal**.



Cette prop est déjà puissante car elle peut prendre JSX de n'importe quelle taille. Dans cet exemple, vous n'avez ajouté qu'un seul élément, mais vous avez pu passer autant de JSX que vous le vouliez. Il n'est pas non plus nécessaire que ce soit JSX. Si vous avez un balisage compliqué par exemple, vous ne voudrez pas le passer directement dans la prop ; ce serait difficile à lire. Au lieu de cela, vous pourriez créer un composant séparé et le faire passer comme prop.

Pour voir comment cela fonctionne, passez AnimalDetails à la prop details :

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

import React from 'react';

...

export default function AnimalCard({ name, size, ...props }) {

return(

<Card

title="Animal"

details={

<AnimalDetails

{...props}

/>

}

>

<h3>{name}</h3>

<div>{size}kg</div>

</Card>

)

}

...

Copy

AnimalDetails est plus compliqué et comporte un certain nombre de lignes de balisage. Si vous deviez l'ajouter directement aux details, cela augmenterait considérablement la prop et la rendrait difficile à lire.

Enregistrez et fermez le fichier. Lorsque vous le ferez, le navigateur se rafraîchira et les détails apparaîtront en haut de la carte.



Vous avez maintenant un composant Card qui peut prendre du JSX personnalisé et le placer à plusieurs endroits. Vous n'êtes pas limité à une seule prop ; vous pouvez passer des éléments à autant de props que vous le souhaitez. Cela vous donne la possibilité de créer des composants wrapper flexibles qui peuvent donner à d'autres développeurs la possibilité de personnaliser un composant tout en conservant son style et sa fonctionnalité d'ensemble.

Faire passer un composant comme prop n'est pas parfait. C'est un peu plus difficile à lire et n'est pas aussi clair que de faire passer children, mais c'est tout aussi flexible et vous pouvez en utiliser autant que vous le souhaitez dans un composant. Vous devez d'abord utiliser children, mais n'hésitez pas à vous rabattre sur les props si cela ne suffit pas.

Dans cette étape, vous avez appris à passer les composants JSX et React comme props à un autre composant. Cela donnera à votre composant la flexibilité nécessaire pour gérer de nombreuses situations où un composant wrapper peut avoir besoin de plusieurs props pour manipuler JSX ou des composants.

Conclusion

Vous avez créé une variété de composants wrapper qui peuvent afficher les données de manière flexible tout en gardant un aspect et une structure prévisibles. Vous avez créé des composants qui peuvent collecter et passer des props inconnues à des composants imbriqués. Vous avez également utilisé la prop children intégrée pour créer des composants happer qui peuvent gérer un nombre arbitraire d'éléments imbriqués. Enfin, vous avez créé un composant qui peut prendre des composants JSX ou React comme props afin que votre composant wrapper puisse gérer plusieurs instances de personnalisations différentes.

Les composants Wrapper vous donnent la possibilité de vous adapter à des circonstances inconnues tout en maximisant la réutilisation et la cohérence du code. Ce modèle est utile pour créer des éléments de base de l'interface utilisateur que vous réutiliserez dans toute une application, notamment : boutons, alertes, modales, diaporamas, etc. Vous y reviendrez souvent.

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