LEARN REACT › QUICK START ›

Thinking in React

React can change how you think about the designs you look at and the apps you build. When you build a user interface with React, you will first break it apart into pieces called *components.* Then, you will describe the different visual states for each of your components. Finally, you will connect your components together so that the data flows through them. In this tutorial, we'll guide you through the thought process of building a searchable product data table with React.

Start with the mockup

Imagine that you already have a JSON API and a mockup from a designer.

The JSON API returns some data that looks like this:

```
[
  { category: "Fruits", price: "$1", stocked: true, name: "Apple" },
  { category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit" },
  { category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"
  { category: "Vegetables", price: "$2", stocked: true, name: "Spinach" },
  { category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin" }
  { category: "Vegetables", price: "$1", stocked: true, name: "Peas" }
]
```

The mockup looks like this:

To implement a UI in React, you will usually follow the same five steps.

Step 1: Break the UI into a component hierarchy

Start by drawing boxes around every component and subcomponent in the mockup and naming them. If you work with a designer, they may have already named these components in their design tool. Check in with them!

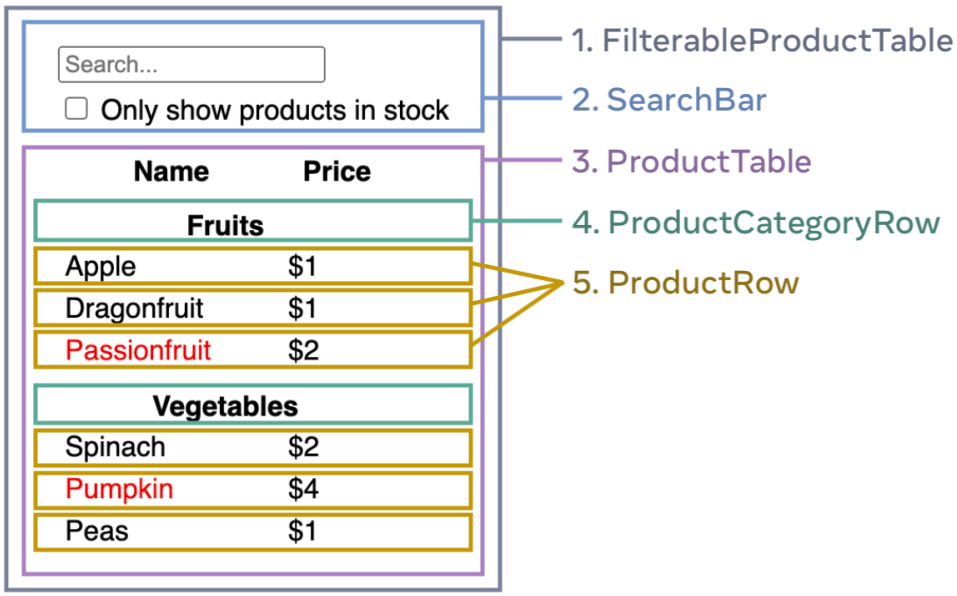
Depending on your background, you can think about splitting up a design into components in different ways:

- **Programming**—use the same techniques for deciding if you should create a new function or object. One such technique is the single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.
- **CSS**—consider what you would make class selectors for. (However, components are a bit less granular.)
- **Design**—consider how you would organize the design's layers.

If your JSON is well-structured, you'll often find that it naturally maps to the

component structure of your UI. That's because UI and data models often have the same information architecture—that is, the same shape. Separate your UI into components, where each component matches one piece of your data model.

There are five components on this screen:



1. `FilterableProductTable` (grey) contains the entire app.
2. `SearchBar` (blue) receives the user input.
3. `ProductTable` (lavender) displays and filters the list according to the user input.
4. `ProductCategoryRow` (green) displays a heading for each category.
5. `ProductRow` (yellow) displays a row for each product.

If you look at `ProductTable` (lavender), you'll see that the table header (containing the "Name" and "Price" labels) isn't its own component. This is a matter of preference, and you could go either way. For this example, it is a part of `ProductTable` because it appears inside the `ProductTable`'s list. However, if this header grows to be complex (e.g., if you add sorting), it would make sense to make this its own `ProductTableHeader` component.

Now that you've identified the components in the mockup, arrange them into a hierarchy. Components that appear within another component in the mockup should appear as a child in the hierarchy:

- `FilterableProductTable`
  - `SearchBar`
  - `ProductTable`
    - `ProductCategoryRow`
    - `ProductRow`

Step 2: Build a static version in React

Now that you have your component hierarchy, it's time to implement your app. The most straightforward approach is to build a version that renders the UI from your data model without adding any interactivity… yet! It's often easier to build the static version first and then add interactivity separately. Building a static version requires a lot of typing and no thinking, but adding interactivity requires a lot of thinking and not a lot of typing.

To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using props. Props are a way of passing data from parent to child. (If you're familiar with the concept of state, don't use state at all to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don't need it.)

You can either build "top down" by starting with building the components higher up in the hierarchy (like `FilterableProductTable`) or "bottom up" by working from components lower down (like `ProductRow`). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up.

App.js Download Reset

App.js Download Reset

```
function ProductCategoryRow({ category }) {
```

```
}

function ProductTable({ products }) {
  const rows = [];
  let lastCategory = null;

  products.forEach((product) => {
    if (product.category !== lastCategory) {
      rows.push(
        <ProductCategoryRow
          category={product.category}
          key={product.category} />
      );
    }
    rows.push(
      <ProductRow
        product={product}
```

```
        product={product}
        key={product.name} />
    );
    lastCategory = product.category;
  });

  return (
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Price</th>
        </tr>
      </thead>
      <tbody>{rows}</tbody>
    </table>
  );
}

function SearchBar() {
  return (
    <form>
      <input type="text" placeholder="Search..." />
      <label>
        <input type="checkbox" />
        {' '}
        Only show products in stock
      </label>
    </form>
  );
}

function FilterableProductTable({ products }) {
  return (
    <div>
      <SearchBar />
      <ProductTable products={products} />
    </div>
  );
}
```

```
80  const PRODUCTS = [
81    {category: "Fruits", price: "$1", stocked: true, name: "Apple"},
82    {category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"}
83    {category: "Fruits", price: "$2", stocked: false, name: "Passionfruit
84    {category: "Vegetables", price: "$2", stocked: true, name: "Spinach"}
85    {category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin'
86    {category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
87  ];
88
89  export default function App() {
90    return <FilterableProductTable products={PRODUCTS} />;
91  }
92
```

Show less

(If this code looks intimidating, go through the Quick Start first!)

After building your components, you'll have a library of reusable components that render your data model. Because this is a static app, the components will only return JSX. The component at the top of the hierarchy (`FilterableProductTable`) will take your data model as a prop. This is called

one-way data flow because the data flows down from the top-level component to the ones at the bottom of the tree.

> **Pitfall**
>
> At this point, you should not be using any state values. That's for the next step!

Step 3: Find the minimal but complete representation of UI state

To make the UI interactive, you need to let users change your underlying data model. You will use *state* for this.

Think of state as the minimal set of changing data that your app needs to remember. The most important principle for structuring state is to keep it DRY (Don't Repeat Yourself). Figure out the absolute minimal representation of the state your application needs and compute everything else on-demand. For example, if you're building a shopping list, you can store the items as an array in state. If you want to also display the number of items in the list, don't store the number of items as another state value—instead, read the length of your array.

Now think of all of the pieces of data in this example application:

1. The original list of products
2. The search text the user has entered
3. The value of the checkbox
4. The filtered list of products

Which of these are state? Identify the ones that are not:

- Does it **remain unchanged** over time? If so, it isn't state.
- Is it **passed in from a parent** via props? If so, it isn't state.
- **Can you compute it** based on existing state or props in your component? If so, it *definitely* isn't state!

What's left is probably state.

Let's go through them one by one again:

1. The original list of products is **passed in as props, so it's not state.**
2. The search text seems to be state since it changes over time and can't be computed from anything.
3. The value of the checkbox seems to be state since it changes over time and can't be computed from anything.
4. The filtered list of products **isn't state because it can be computed** by taking the original list of products and filtering it according to the search text and value of the checkbox.

This means only the search text and the value of the checkbox are state!

Nicely done!

DEEP DIVE

Props vs State

Hide Details

There are two types of "model" data in React: props and state. The two are very different:

- **Props** are like arguments you pass to a function. They let a parent component pass data to a child component and customize its appearance. For example, a `Form` can pass a `color` prop to a `Button`.
- **State** is like a component's memory. It lets a component keep track of some information and change it in response to interactions. For example, a `Button` might keep track of `isHovered` state.

Props and state are different, but they work together. A parent component will often keep some information in state (so that it can change it), and *pass it down* to child components as their props. It's okay if the difference still feels fuzzy on the first read. It takes a bit of practice for it to really stick!

Step 4: Identify where your state should live

After identifying your app's minimal state data, you need to identify which component is responsible for changing this state, or *owns* the state. Remember: React uses one-way data flow, passing data down the component hierarchy from parent to child component. It may not be immediately clear which component should own what state. This can be challenging if you're new to this concept, but you can figure it out by following these steps!

For each piece of state in your application:

1. Identify *every* component that renders something based on that state.
2. Find their closest common parent component—a component above them all in the hierarchy.
3. Decide where the state should live:
    1. Often, you can put the state directly into their common parent.
    2. You can also put the state into some component above their common parent.

parent.

3. If you can't find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common parent component.

In the previous step, you found two pieces of state in this application: the search input text, and the value of the checkbox. In this example, they always appear together, so it is easier to think of them as a single piece of state.

Now let's run through our strategy for this state:

1. **Identify components that use state:**
   - `ProductTable` needs to filter the product list based on that state (search text and checkbox value).
   - `SearchBar` needs to display that state (search text and checkbox value).
2. **Find their common parent:** The first parent component both components share is `FilterableProductTable`.
3. **Decide where the state lives**: We'll keep the filter text and checked state values in `FilterableProductTable`.

So the state values will live in `FilterableProductTable`.

Add state to the component with the `useState()` Hook. Hooks let you "hook into" a component's render cycle. Add two state variables at the top of `FilterableProductTable` and specify the initial state of your application:

```
function FilterableProductTable({ products }) {
  const [filterText, setFilterText] = useState('');
  const [inStockOnly, setInStockOnly] = useState(false);
```

Then, pass `filterText` and `inStockOnly` to `ProductTable` and `SearchBar` as props:

```
<div>
  <SearchBar
    filterText={filterText}
    inStockOnly={inStockOnly} />
  <ProductTable
    products={products}
    filterText={filterText}
    inStockOnly={inStockOnly} />
</div>
```

You can start seeing how your application will behave. Edit the `filterText` initial value from `useState('')` to `useState('fruit')` in the sandbox code below. You'll see both the search input text and the table update:

App.js Download Reset

```
import { useState } from 'react';

function FilterableProductTable({ products }) {
  const [filterText, setFilterText] = useState('');
  const [inStockOnly, setInStockOnly] = useState(false);

  return (
    <div>
      <SearchBar
        filterText={filterText}
        inStockOnly={inStockOnly} />
      <ProductTable
        products={products}
        filterText={filterText}
        inStockOnly={inStockOnly} />
    </div>
  );
}

function ProductCategoryRow({ category }) {
  return (
    <tr>
```

```
    <tr>
      <th colSpan="2">
        {category}
      </th>
    </tr>
  );
}

function ProductRow({ product }) {
  const name = product.stocked ? product.name :
    <span style={{ color: 'red' }}>
      {product.name}
    </span>;

  return (
    <tr>
      <td>{name}</td>
      <td>{product.price}</td>
    </tr>
  );
}

function ProductTable({ products, filterText, inStockOnly }) {
  const rows = [];
  let lastCategory = null;

  products.forEach((product) => {
    if (
      product.name.toLowerCase().indexOf(
        filterText.toLowerCase()
      ) === -1
    ) {
      return;
    }
    if (inStockOnly && !product.stocked) {
      return;
    }
    if (product.category !== lastCategory) {
      rows.push(
        <ProductCategoryRow
          category={product.category}
```

```
62            category={product.category}
63            key={product.category} />
```

Show less

Console (2)

```
Warning: You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.
input
form
SearchBar@https://dad0ba0e.sandpack-bundler-4bw.pages.dev/App.js:122:7
```

Notice that editing the form doesn't work yet. There is a console error in the sandbox above explaining why:

> Console

```
You provided a `value` prop to a form field without an
`onChange` handler. This will render a read-only field.
```

In the sandbox above, `ProductTable` and `SearchBar` read the `filterText` and `inStockOnly` props to render the table, the input, and the checkbox. For example, here is how `SearchBar` populates the input value:

```
function SearchBar({ filterText, inStockOnly }) {
  return (
    <form>
      <input
        type="text"
        value={filterText}
        placeholder="Search..."/>
```

However, you haven't added any code to respond to the user actions like typing yet. This will be your final step.

Step 5: Add inverse data flow

Currently your app renders correctly with props and state flowing down the hierarchy. But to change the state according to user input, you will need to support data flowing the other way: the form components deep in the hierarchy need to update the state in `FilterableProductTable`.

React makes this data flow explicit, but it requires a little more typing than two-way data binding. If you try to type or check the box in the example above, you'll see that React ignores your input. This is intentional. By writing `<input value={filterText} />`, you've set the `value` prop of the `input` to always be equal to the `filterText` state passed in from `FilterableProductTable`. Since `filterText` state is never set, the input never changes.

You want to make it so whenever the user changes the form inputs, the state updates to reflect those changes. The state is owned by `FilterableProductTable`, so only it can call `setFilterText` and `setInStockOnly`. To let `SearchBar` update the `FilterableProductTable`'s state, you need to pass these functions down to `SearchBar`:

```
function FilterableProductTable({ products }) {
  const [filterText, setFilterText] = useState('');
  const [inStockOnly, setInStockOnly] = useState(false);

  return (
    <div>
      <SearchBar
        filterText={filterText}
        inStockOnly={inStockOnly}
        onFilterTextChange={setFilterText}
        onInStockOnlyChange={setInStockOnly} />
```

Inside the `SearchBar`, you will add the `onChange` event handlers and set the parent state from them:

```
<input
  type="text"
  value={filterText}
  placeholder="Search..."
  onChange={(e) => onFilterTextChange(e.target.value)} />
```

Now the application fully works!

App.js Download Reset

```
import { useState } from 'react';
```

```
  onInStockOnlyChange
}) {
  return (
    <form>
      <input
        type="text"
        value={filterText} placeholder="Search..."
        onChange={(e) => onFilterTextChange(e.target.value)} />
      <label>
        <input
          type="checkbox"
          checked={inStockOnly}
          onChange={(e) => onInStockOnlyChange(e.target.checked)} />
        {' '}
        Only show products in stock
      </label>
    </form>
  );
}

const PRODUCTS = [
  {category: "Fruits", price: "$1", stocked: true, name: "Apple"},
  {category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"
  {category: "Fruits", price: "$2", stocked: false, name: "Passionfrui
  {category: "Vegetables", price: "$2", stocked: true, name: "Spinach"
  {category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin
  {category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];

export default function App() {
```

```
123    return <FilterableProductTable products={PRODUCTS} />;
124  }
125
```

Show less

You can learn all about handling events and updating state in the Adding Interactivity section.

Where to go from here

This was a very brief introduction to how to think about building components and applications with React. You can start a React project right now or dive deeper on all the syntax used in this tutorial.

PREVIOUS
Tutorial: Tic-Tac-Toe

How do you like these docs?

Take our survey!

FACEBOOK

Open Source

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

Acknowledgements

Docs Contributors

Meet the Team

Blog

More

React Native

Privacy

Terms