An introduction to React

What to expect

- A 6-8 hour interactive workshop
- Learning about React while building something with it
- Using modern React (functional components)
- Code examples and try-it-yourself challenges
- Feel free to ask questions

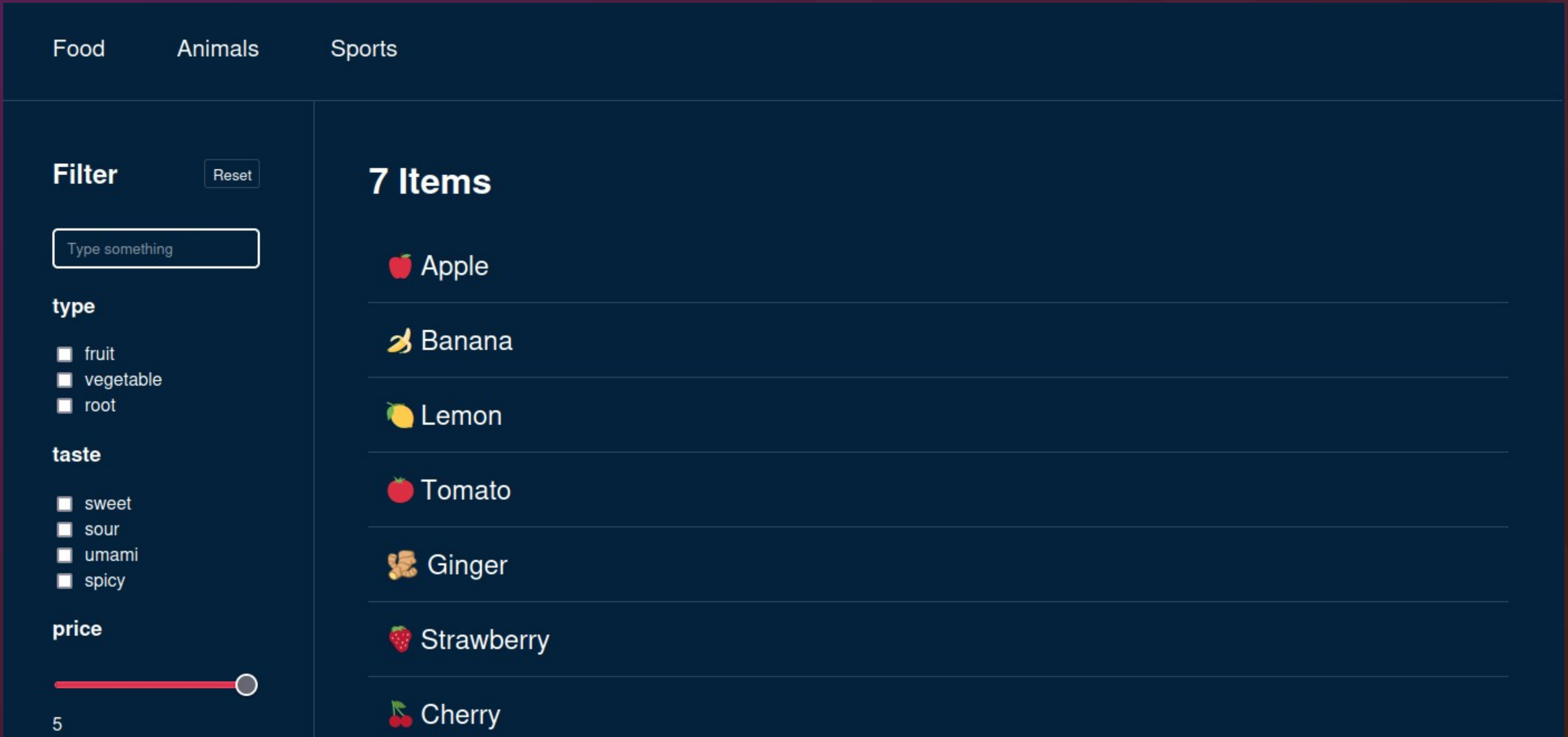
What not to expect

- Full coverage of every React feature (time)
- Server-side or react-native code (mobile)
- In-depth look at frameworks and libraries

Questions?

Let’s build a filter-app together!

Example: Final product (end goal)



7 Items

🍎 Apple

🍌 Banana

🍋 Lemon

🍅 Tomato

🫚 Ginger

🍓 Strawberry

🍒 Cherry

Why React?

- An industry standard
- Well documented, matured library
- Many 3rd party packages & tools
- Reusable components
- State management

Quick setup with Vite

1.

```
npm create vite@latest
```

2.

```
Need to install the following packages:
  create-vite@4.4.1
Ok to proceed? (y) y
✔️ Project name: … vite-project
? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
    Vue
❯   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
```

```
? Select a variant: › - Use arrow-keys. Return to submit.
    TypeScript
❯   TypeScript + SWC
    JavaScript
    JavaScript + SWC
```

3.

```
npm install
```

```
npm run dev
```

Where to start?

- Start with a single component
- Make it return basic & static HTML
- Future steps:
  - Make it dynamic (components & properties)
  - Make it interactive (state & event-handlers)

HTML structure within `.jsx`

- Conditionals
- Iterations
- Fragments

Conditionals

```
<div>
    {isLoading && <p>Loading...</p>}
</div>
```

```
<div>
    {isLoading ? <p>Loading...</p> : <Filter />}
</div>
```

- JavaScript expressions can be put inside JSX by placing it between {}
- There are many strategies for conditional JSX

Iterations

```
{options.map((option) => (
    <li key={option.id}>
        {option.id}
    </li>
))}
```

Arrays can be iterated over and each return more JSX

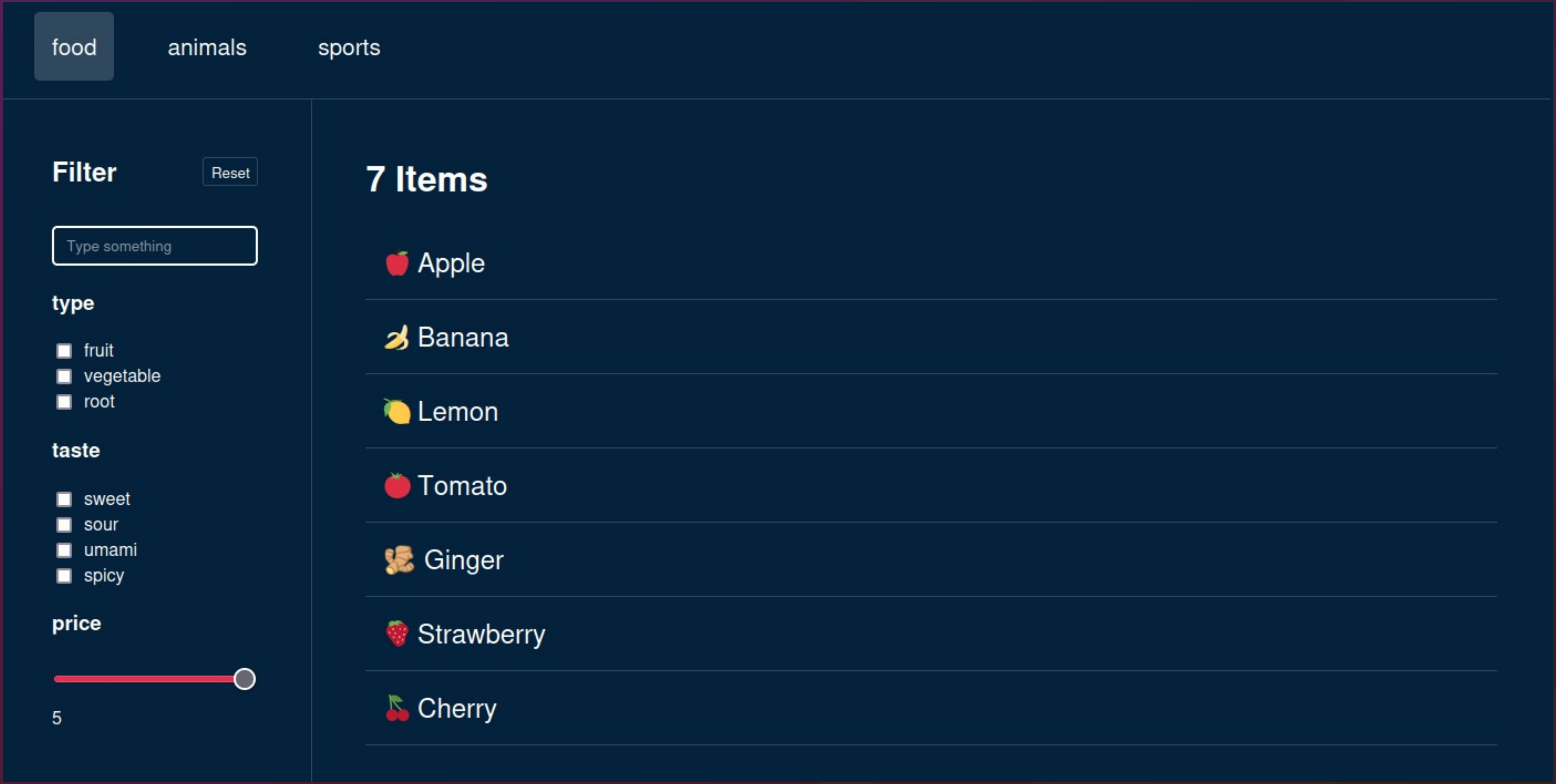
Fragments

```
<>
    <h1>Hello</h1>
    <p>World</p>
</>
```
✅

```
<Fragment>
    <h1>Hello</h1>
    <p>World</p>
</Fragment>
```
✅

```
<div>
    <h1>Hello</h1>
    <p>World</p>
</div>
```
❌

- React requires single top-level elements
- Fragments group items without affecting the DOM (`<div>` could affect CSS & JS selectors)


food
animals
sports

Filter
Reset

Type something

type

- fruit
- vegetable
- root

taste

- sweet
- sour
- umami
- spicy

price

5

7 Items

🍎 Apple

🍌 Banana

🍋 Lemon

🍅 Tomato

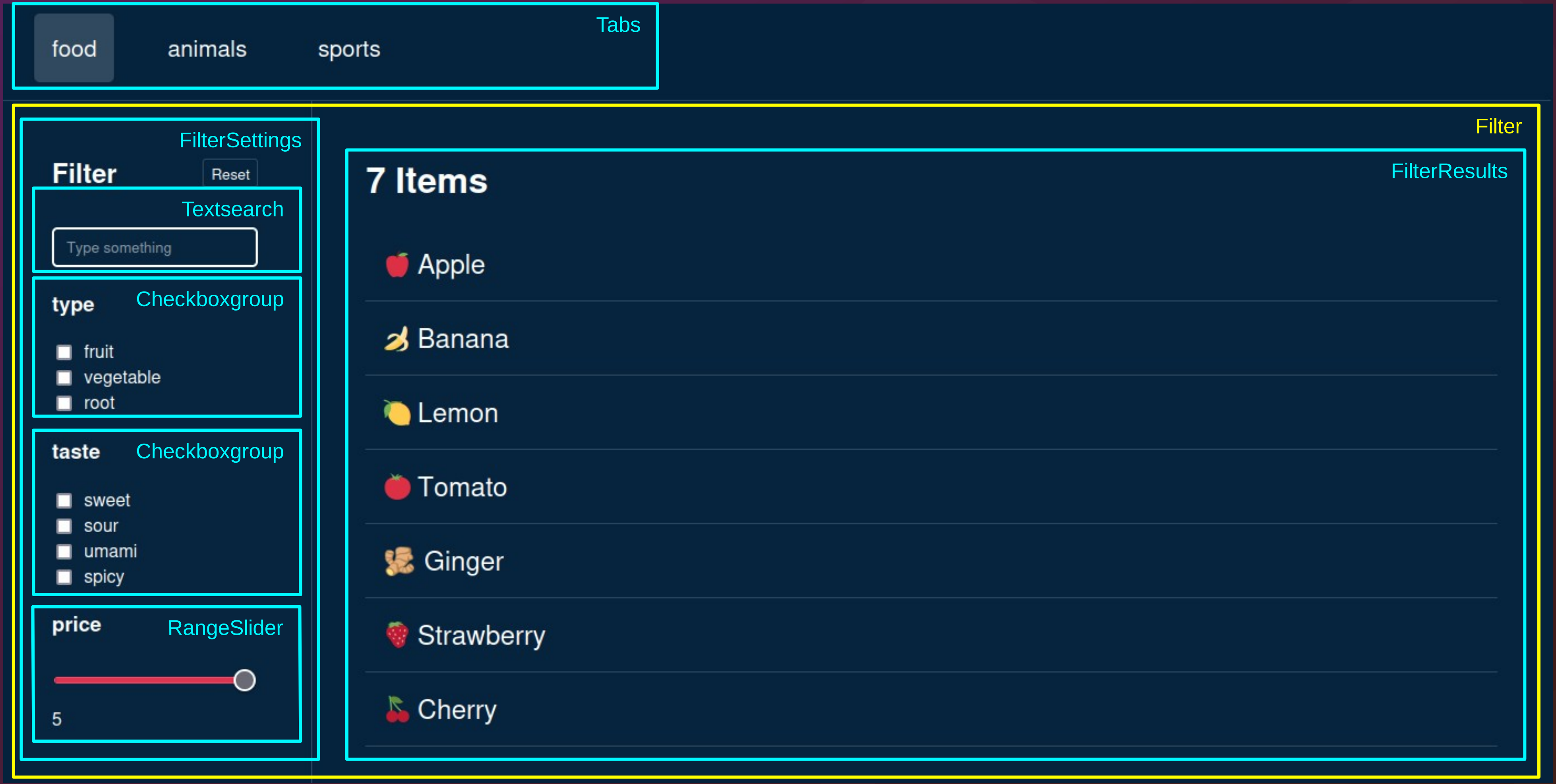
🫚 Ginger

🍓 Strawberry

🍒 Cherry

Split code into components

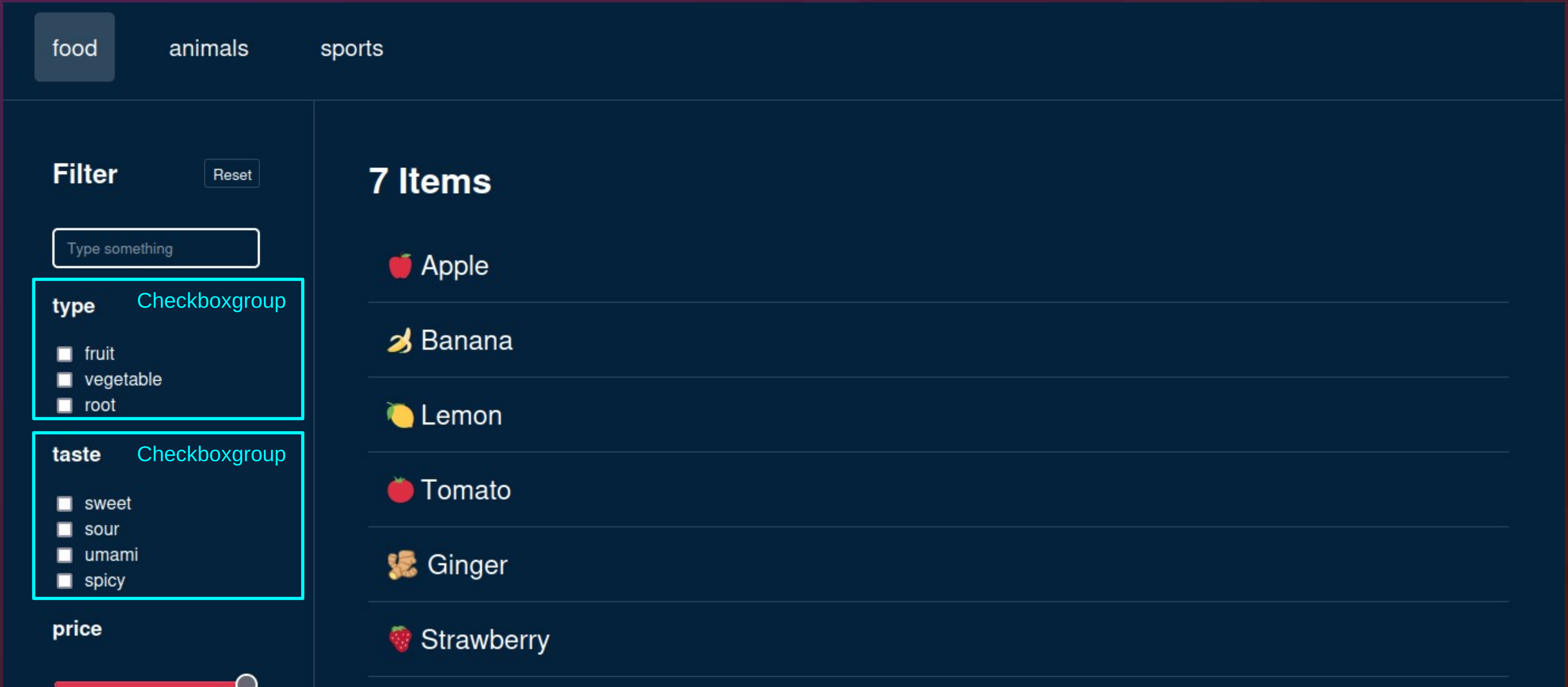
- Reusable
- Configurable
- Testable
- Readable



Example: Splitting into components

Dynamic components

Example: Splitting into components



Component properties

- Make components re-usable
- Hands over state to parent components

Component properties

```
<Checkboxgroup
    label="taste"
    options={["sweet", "sour", "umami"]}
/>
```

```
<Checkboxgroup
    label="type"
    options={["fruit", "vegetables"]}
/>
```

Example: Component configuration
components/FilterSettings.tsx

```
export function Checkboxgroup({ label, options }) {
    return (
        <>
            <h3>{label}</h3>
            <ul>
                {options.map((option) => (
                    <li key={option}>
                        <label>
                            <input
                                name={`${label}_checkboxgroup`}
                                type="checkbox"
                                value={option}
                            />
                            {option}
                        </label>
                    </li>
                ))}
            </ul>
        </>
    );
}
```

Example: Component configuration
components/Checkboxgroup.tsx

👉Interactivity

User input 🖱️⌨️📱🕹️

Visually our app looks finished, but it lacks interactivity

For now let’s focus on direct user events

Examples: click, scroll, focus, input,..

Event-handling

```
document.querySelector('#search').addEventListener('change', (e) => {
    updateFilterResults(e.target.value);
});
```

❌

This is problematic because:

- React’s virtual DOM frequently recreates nodes
- Every render would create additional event-handlers

React event-handling

```
<input
    type="search"
    value={value}
    onChange={(event) => {/* do something */}}
/>
```

```
<button onClick={(event) => /* do something */}>
     Reset
</button>
```

- Attach events directly inside .jsx
- Event-handlers change state, which is defined outside of .jsx

📃 Interactivity checklist:

⬜ Create a variable to hold state

⬜ Use the state in our component (JSX & JS)

⬜ Update state on user-input (event)

⬜ Re-render the app after state updates

State

🪝 useState()

```
const [activeTab, setActiveTab] = useState(0);
const [search, setSearch] = useState("");
```

```
<input type="search"
     value={search}
     onChange={e => setSearch(e.target.value)}
/>
```

- Returns a state variable and a set function
- State is saved between renders
- React will re-render when state changes

Interactivity checklist

✅ Create a variable to hold state

```
const [search, setSearch] = useState("");
```

✅ Use the state in our component (JSX & JS)

```
<input type="search" value={search} onChange={e => setSearch(e.target.value)} />
```

✅ Update state on user-input (event)

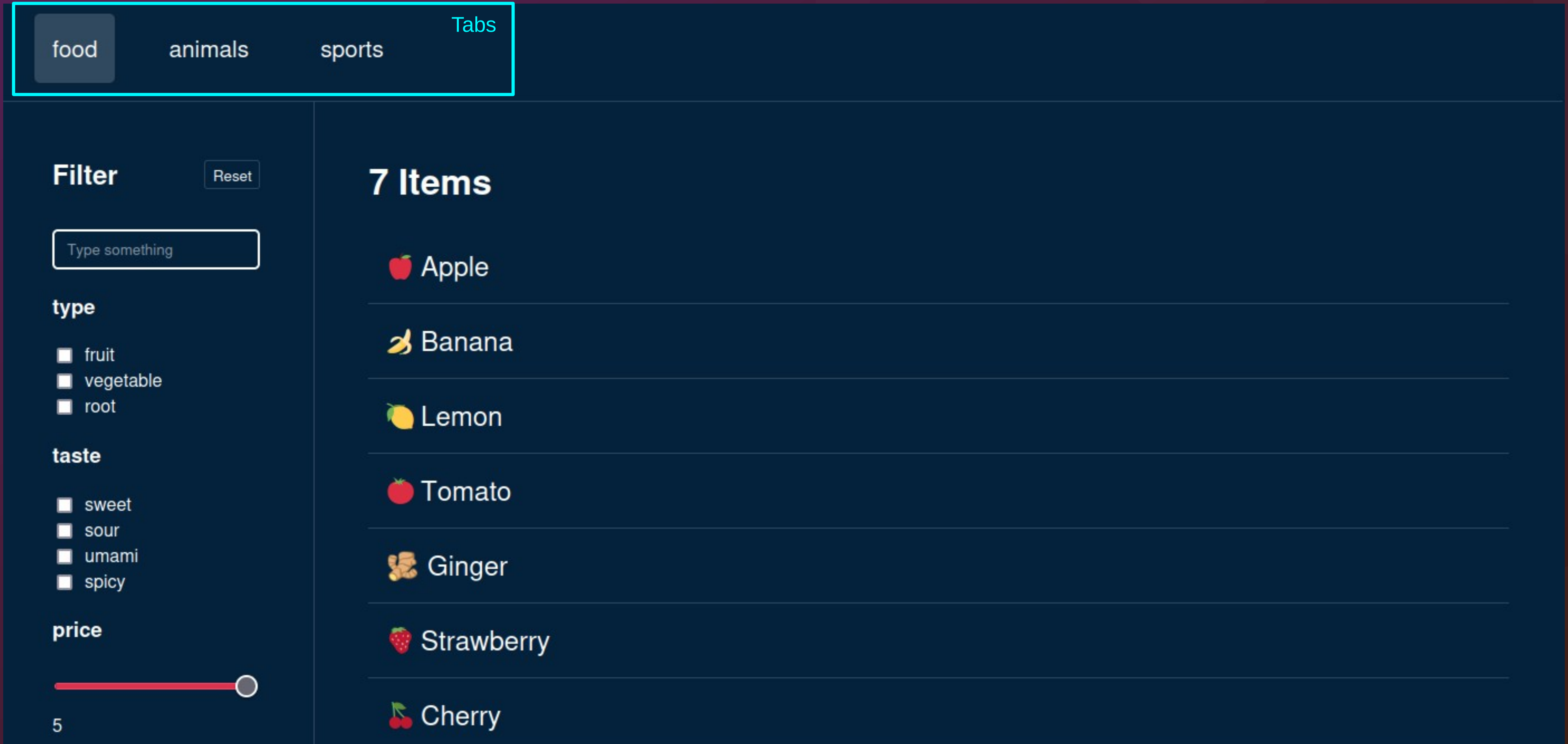
```
<input type="search" value={search} onChange={e => setSearch(e.target.value)} />
```

✅ Re-render the app after state updates

```
// React re-renders automatically after state mutates!
```

Interactive <Tabs>

Example: Interactive <Tabs>



Tabs

food animals sports

Filter Reset

Type something

type

- fruit
- vegetable
- root

taste

- sweet
- sour
- umami
- spicy

price

5

7 Items

- 🍎 Apple
- 🍌 Banana
- 🍋 Lemon
- 🍅 Tomato
- 🫚 Ginger
- 🍓 Strawberry
- 🍒 Cherry

Interactive <Tabs>

```tsx
import { useState } from "react";
import { Tabs } from "./components/Tabs";

export function ExampleInteractiveTabs() {
  const options = ["food", "animals", "sports"];
  const [activeFilter, setActiveFilter] =
  useState(options[0]);

  return (
    <>
      <Tabs
        options={options}
        active={activeFilter}
        onUpdate={setActiveFilter}
      />
      <pre>Checked tab: {activeFilter}</pre>
    </>
  );
}
```

Example: Interactive <Tabs>
InteractiveTabs.tsx

```tsx
export function Tabs({ options, active, onUpdate }) {
  return (
    <div className="filter-navigation" role="tablist">
      {options.map((option) => (
        <button
          key={option}
          role="tab"
          aria-selected={active === option}
          className={`filter-navigation__button${
            active === option ? " active" : ""
          }`}
          onClick={() => onUpdate(option)}
        >
          {option}
        </button>
      ))}
    </div>
  );
}
```

Example: Interactive <Tabs>
components/Tabs.tsx

let, const, var

```
const search = “”

<input type="search"
     value={search}
     onChange={e => { search = e.target.value }}
/>
```

❌

Wouldn’t work because:

- The variable gets recreated on every render
- Updating the variable won’t tell react to re-render

`let, const, var`

```
const [search, setSearch] = useState("");
const searchLowerCase = search.toLowerCase();
```

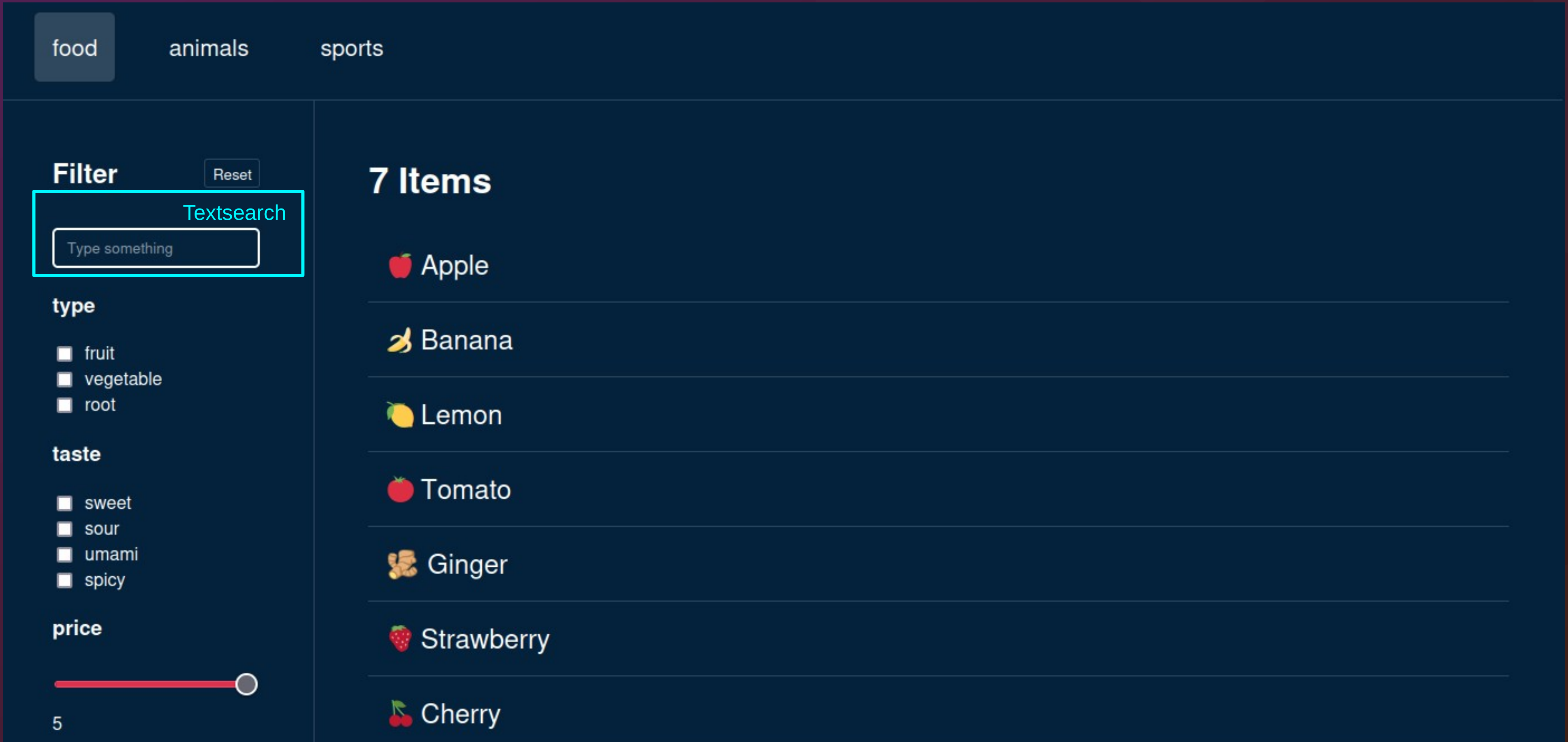
- Not everything needs to be defined with `useState()`
- For example: derived variables
  - They won’t need to be changed directly
  - Their origin will already cause a re-render on change

Lifting state up

- In React data always flows from the top to the bottom
- If state needs to be shared between adjacent components, move it to a parent and pass it down with properties
- Pass `setState` to a child so it can change the state of a parent, from where the updated state will flow down afterwards

Interactive <TextSearch>

Example: Interactive <TextSearch>



food animals sports

Filter Reset

Textsearch

Type something

type

- ☐ fruit
- ☐ vegetable
- ☐ root

taste

- ☐ sweet
- ☐ sour
- ☐ umami
- ☐ spicy

price

5

7 Items

- 🍎 Apple
- 🍌 Banana
- 🍋 Lemon
- 🍅 Tomato
- 🫚 Ginger
- 🍓 Strawberry
- 🍒 Cherry

Interactive <TextSearch>

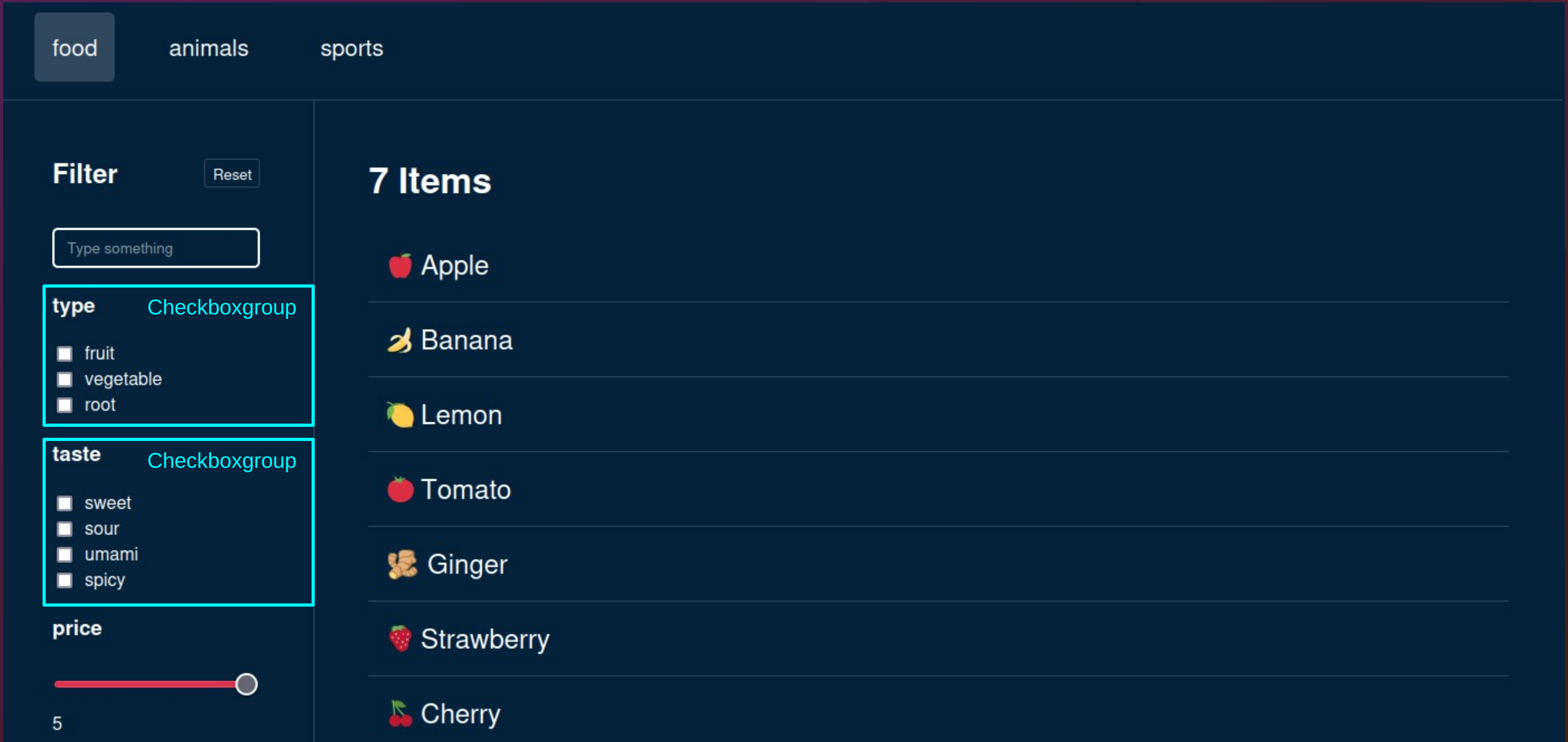
```
import { useState } from "react";
import { Tabs } from "./components/Tabs";

export function ExampleTabs() {
  const options = ["food", "animals", "sports"];
  const [activeFilter, setActiveFilter] =
  useState(options[0]);

  return (
    <>
      <Tabs
        options={options}
        active={activeFilter}
        onUpdate={setActiveFilter}
      />
      <pre>Checked tab: {activeFilter}</pre>
    </>
  );
}
```

```
export function Tabs({ options, active, onUpdate }) {
  return (
    <div className="filter-navigation" role="tablist">
      {options.map((option) => (
        <button
          key={option}
          role="tab"
          aria-selected={active === option}
          className={`filter-navigation__button${
            active === option ? " active" : ""
          }`}
          onClick={() => onUpdate(option)}
        >
          {option}
        </button>
      ))}
    </div>
  );
}
```

Work in progress

Interactive <Checkboxgroup>

Example: Interactive <Checkboxgroup>



Interactive <Checkboxgroup>

```jsx
import { useState } from "react";
import { Tabs } from "./components/Tabs";

export function ExampleTabs() {
  const options = ["food", "animals", "sports"];
  const [activeFilter, setActiveFilter] =
  useState(options[0]);

  return (
    <>
      <Tabs
        options={options}
        active={activeFilter}
        onUpdate={setActiveFilter}
      />
      <pre>Checked tab: {activeFilter}</pre>
    </>
  );
}
```

```jsx
export function Tabs({ options, active, onUpdate }) {
  return (
    <div className="filter-navigation" role="tablist">
      {options.map((option) => (
        <button
          key={option}
          role="tab"
          aria-selected={active === option}
          className={`filter-navigation__button${
            active === option ? " active" : ""
          }`}
          onClick={() => onUpdate(option)}
        >
          {option}
        </button>
      ))}
    </div>
  );
}
```

Work in progress

Recap

🪝 Hooks

useState

useRef

useSyncExternalStore

useCallback

useTransition

useEffect

useInsertionEffect

useContext

useReducer

useImperativeHandle

useLayoutEffect

useDeferredValue

use

useId

useDebugValue

useMemo

🪝 Hooks

- Hook are functions that let you “hook” into React’s state
- They need to run
  - inside a functional component
  - in the same order every render
  - unconditionally

Pure Functions inside React

- Always produce same output with same input
- Have no side-effects

Functional Components

- React can easily abort incomplete renders
- Results can be cached more easily
- Components can be rendered on the server

Mutating state

- Inside event handlers
- Inside `useEffect()`

Detecting state changes

We want to fetch our data when the app initializes and when the category is changed

🪝useEffect()

- Run after the component has rendered
- Has a dependency array that needs to match the effects dependencies
- If dependencies are empty it only runs once
- The returned function runs when component unmounts

🪝useEffect()

- Its purpose is to synchronize with external systems
- Send analytics events
- Connect to a native (video) or external API (image gallery)

🪝useEffect()

Can easily be misused and introduce unnecessary complexity and performance costs.

How to avoid prop drilling?

Filter.tsx

```
<div className="filter">
    <FilterSettings
        filters={filters}
        updateCategory={updateCategory}
        updateRange={updateRange}
        updateSearch={updateSearch}
        reset={reset}
    />
    <FilterResults
        items={filterResults}
        search={filters.search.value}
        reset={reset}
    />
</div>
```

FilterSettings.tsx

```
<div className="filter__settings">
    <FilterSettingsToolbar reset={reset} />

    <TextSearch
        value={filters.search.value}
        onUpdate={(e) =>
        updateSearch(e.target.value)}
    />
    ...
</div>
```

🪝 useContext()

- Provides a shared state to all children who subscribe with the useContext hook
- Other components in between don’t have to know the data exists
- Can lead to unintended re-renders

🪝useContext()

Filter.tsx

```
export const FilterContext = createContext(null);
```

```
<FilterContext.Provider items={items} >
    <FilterSettings />
    <FilterResults />
</FilterContext.Provider>
```

FilterSettings.tsx &
FilterResults.tsx

```
import { FilterContext } from './Filter.tsx'
```

```
items = useContext(FilterContext)
```

🪝useRef()

- Changing ref.current doesn’t cause a rerender!

Component updates

- React tries to only update parts of DOM which need to be updated
- React has it’s own virtual DOM
- Sometimes we want to explicitly update components

key attribute

- Whenever the key attribute changes, React will treat it like a different element and cause a rerender.

How to handle complex state

The filtersettings are increasing in complexity and we want a better way than useState to handle it in a unified way.

🪝 useReducer()

- UseReducer() is similar to useState() but it can provide multiple functions to update the state it holds

JS mutations

	copy	mutating
add	.concat(), [...arr]	.push(), .unshift()
remove	.filter(), slice()	.pop(), .shift(), .splice()
replace	.map()	.splice(), arr[i] =
sort	[...arr] => arr.sort()	.reverse(), .sort()

Source: https://react.dev/learn/updating-arrays-in-state

Copying deeply nested state

Copying nested state can become very tedious and hard to read:

```
case "SET_RANGE":
    return {
    ...state,
    ranges: {
        ...state.ranges,
        [action.payload.id]: {
            ...state.ranges[action.payload.id],
            value: action.payload.value,
        },
    },
};
```

useFilter.tsx:41

Immer (Framework)

- Immer allows state to be mutated directly
  - Replace `useState()` with `useImmer()`
  - Replace `useReducer()` with `immerReducer()`

```
case "SET_RANGE":
    return {
    ...state,
    ranges: {
        ...state.ranges,
        [action.payload.id]: {
            ...state.ranges[action.payload.id],
            value: action.payload.value,
        },
    },
};
```

→

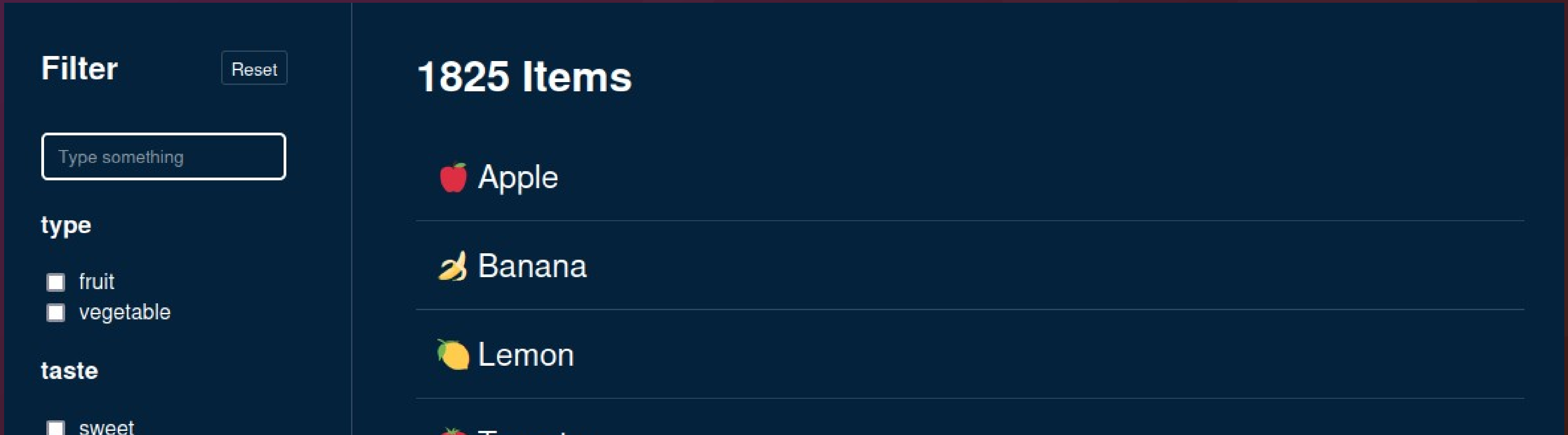
```
case "SET_RANGE":
    state.ranges[action.payload.id].value =
    action.payload.value;
    break;
```

🪝 Custom hooks

- Need to start with use
- Only are considered hooks if they wrap around React hooks, otherwise they’re just functions
- Are essentially just wrappers like functions

⚠️ Performance issues

If we had complex filters with thousands of items, slower devices might run into performance issues


Filter Reset

Type something

type

- fruit
- vegetable

taste

- sweet

1825 Items

🍎 Apple

🍌 Banana

🍋 Lemon

🪝 useMemo()

```
const filterResults = useMemo(() => {
// costly filter operation

return filtereditems;
}, [filters, items]);
```

useFilterContext.tsx:30

- Cache the result of a heavy computation
- Not needed for simple calculations
- Only updates when dependency array changes

🪝 `useCallback()`

- Like `useMemo()` but it returns a function instead of a value
- Referential equality

Typescript

State Management

- SWR
- React Query
- Redux & Redux Toolkit
- Zustand
- Jotai

Routing

- React-Router
- NextJS

Next steps & Ideas

- Add routing for detailviews
- Save the filter-state in url for deep linking
- Extract Filter types into separate modules

Recommendations

- Official React.dev - Learn & Documentation

  https://react.dev/learn

- Jack Herrington - Typescript & React

  https://www.youtube.com/watch?v=j8AVXNozac8

- Web Dev Simplified – React Hooks Explained

  https://www.youtube.com/watch?v=O6P86uwfdR0

Thank you!