

An introduction to React

A large, faint, light blue watermark of the React logo is centered in the background of the slide. The logo consists of a stylized 'R' formed by two interlocking loops.

What to expect

- Hands on approach from the view of a developer
- A journey through what React has to offer
- Some Typescript
- An interactive experience

What not to expect

- A full in-depth explanation of every single feature
- Frameworks and Libraries

Let's build a filter-app together!

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

Why React?

- A virtual DOM implementation
- Handle state

Quick setup with Vite

- Create-react-app
- Vite
- NextJS

HTML structure with JSX

- Conditionals
- Iterations
- Fragments
- TSX for Typescript

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

```
<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

Fragments

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



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



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



- React sometimes requires single top-level elements
- Fragments group items together 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

food

animals

sports

Tabs

FilterSettings

Filter

Reset

Textsearch

Type something

type

Checkboxgroup

- ☐ fruit
- ☐ vegetable
- ☐ root

taste

Checkboxgroup

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

price

RangeSlider



5


Filter


FilterResults

7 Items


 Apple


 Banana

 Lemon

 Tomato

 Ginger

 Strawberry

 Cherry

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

foodanimalssports

food

animals

sports

Reset

Type something

type


- fruit
- vegetable
- root


taste


- sweet
- sour
- umami
- spicy


price5


7 Items


 Apple


 Banana

 Lemon

 Tomato

 Ginger

 Strawberry

 Cherry

food

animals

sports

Filter

Reset

Type something

type Checkboxgroup

- ☐ fruit
- ☐ vegetable
- ☐ root

taste Checkboxgroup

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

price





5

7 Items

 Apple


 Banana

 Lemon

 Tomato

 Ginger

 Strawberry

 Cherry

Component properties

Ideally this component can be used anywhere

We can build new base-level components like

datepickers, or a simple

`input[type="password"]` and reuse them

throughout our app

👉 Interactivity

Before we can build our reusable components, we need to know how to make them interactive.

👉 Interactivity

```
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

Interactivity checklist:

- ☐ Create a variable
- ☐ Use the variable in our component (JSX & JS)
- ☐ Update the variable on user-input
- ☐ Re-render the app after a variable updates

useState()

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

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

- Allows to save state between renders
- Tells React to re-render when state changes

Interactivity checklist

✓ Create a variable

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

✓ Use the variable in our component (JSX & JS)

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

✓ Update the variable on user-input

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

✓ Re-render the app after a variable updates

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

Let's use our new tools

foodanimalssports

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

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

Hooks

useState

useRef

useSyncExternalStore

useTransition

useCallback

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)
```

useContext()

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



```
<div className="filter">
  <FilterContextProvider items={items}>
    <FilterSettings />
    <FilterResults />
  </FilterContextProvider>
</div>
```

Filter.tsx:8

useRef()

- Changing `ref.current` doesn't cause a rerender!

Bug report

We're updating the items, but the Filter still shows the old data.

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 filter settings 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	<code>.concat(), [...arr]</code>	<code>.push(), .unshift()</code>
remove	<code>.filter(), slice()</code>	<code>.pop(), .shift(), .splice()</code>
replace	<code>.map()</code>	<code>.splice(), arr[i] =</code>
sort	<code>[...arr] => arr.sort()</code>	<code>.reverse(), .sort()</code>

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

 Tomato

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!