

# SENG 365 Week 9

## React: Event Handling, Hooks, Global State





## **This week**

---

- Event handling
- Hooks
- Global storage
- Routing
- Debugging React apps



# Event handling and hooks

---



## Event handling

---

- Event handling in a component is similar to DOM events but with some differences
  - Camel case notation: `onClick`, `onSubmit`, etc.
  - Pass a function as the event handler
- The function can be a method in the component class but cannot be called unless `this` is explicitly bound.



## Digression: function binding

```
const module = {  
  x: 42,  
  getX: function() {  
    return this.x;  
  }  
};  
  
const unboundGetX = module.getX;  
console.log(unboundGetX()); // The function gets invoked at the global scope  
// expected output: undefined  
  
const boundGetX = unboundGetX.bind(module);  
console.log(boundGetX());  
// expected output: 42
```



## Accessing a component method for an event (option 1)

```
class Toggle extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {isToggleOn: true};  
  
    // This binding is necessary to make `this` work in the callback  
    this.handleClick = this.handleClick.bind(this);  
  }  
  
  handleClick() {  
    this.setState(prevState => ({  
      isToggleOn: !prevState.isToggleOn  
    }));  
  }  
  
  render() {  
    return (  
      <button onClick={this.handleClick}>  
        {this.state.isToggleOn ? 'ON' : 'OFF'}  
      </button>  
    );  
  }  
}
```

Bind `this` in the component constructor.



## Accessing a component method for an event (option 2)

```
class LoggingButton extends React.Component {  
  // This syntax ensures `this` is bound within handleClick.  
  // Warning: this is *experimental* syntax.  
  handleClick = () => {  
    console.log('this is:', this);  
  }  
  
  render() {  
    return (  
      <button onClick={this.handleClick}>  
        Click me  
      </button>  
    );  
  }  
}
```

Use public class fields syntax

This syntax is experimental but enabled by default by Create React App



## Accessing a component method for an event (option 3)

```
class LoggingButton extends React.Component {  
  handleClick() {  
    console.log('this is:', this);  
  }  
  
  render() {  
    // This syntax ensures `this` is bound within handleClick  
    return (  
      <button onClick={() => this.handleClick()}>  
        Click me  
      </button>  
    );  
  }  
}
```

Use arrow function in callback

Not as performant because creates new function with each render





## Function components with state

---

- Function components have been around in React for a long time for simple components
- Stateless before React 16.8
- **Hooks** were added in React 16.8
- Now most of what you could do with class components can be done with function components
- Syntax much lighter, easier to read code



## Hooks in Function Components

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Import `useState` hook

`count` is assigned the initial value which is zero.

`setCount` is assigned to a function used to modify the state.

Note: must use arrow function notation, otherwise if we execute `setCount(count + 1)` in the return statement it will get in an endless render loop.



## Hooks in Function Components

```
const App = () => {  
  const [ counter, setCounter ] = useState(0)  
  
  const increaseByOne = () => setCounter(counter + 1)  
  
  const setToZero = () => setCounter(0)  
  
  return (  
    <div>  
      <div>{counter}</div>  
      <button onClick={increaseByOne}>  
        plus  
      </button>  
      <button onClick={setToZero}>  
        zero  
      </button>  
    </div>  
  )  
}
```

If we don't want to use an arrow function in the render, declare a **const** set equal to the arrow function in the component body.

(Note in this example App component function is also using the **const** function expression form, instead of **function** keyword)



## Accessing previous state explicitly in setter function

```
function Counter({ initialCount }) {  
  const [count, setCount] = useState(initialCount);  
  return (  
    <div>  
      Count: {count}  
      <button onClick={() => setCount(initialCount)}>Reset</button>  
      <button onClick={() => setCount((prevCount) => prevCount - 1)}>-  
</button>  
      <button onClick={() => setCount((prevCount) => prevCount + 1)}>+  
</button>  
    </div>  
  );  
}
```



## useState can be used to set multiple state variables

```
function App() {  
  const [sport, setSport] = useState('basketball');  
  const [points, setPoints] = useState(31);  
  const [hobbies, setHobbies] = useState([]);  
}
```

Compare with `this.state` in class components.

Updating a state variable *replaces* the variable, in contrast to merging as in `this.setState()`



## useReducer

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Pass in a reducer of type (state, action) => newState

Returns state and a dispatch method that can be used to trigger different actions.

Useful to maintain state of complex objects

In React source useState is just a special case of the useReducer hook



## useEffect hook

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Creates **side effects** in components

Tells component that it needs to do something *after* render

Guarantees DOM has been updated

Allows similar functionality to lifecycle methods in a class component



## useEffect optimization

Second parameter is a **dependency array**: specifies what state variables must change to execute side effect

```
useEffect(() => {  
  document.title = `You clicked ${count} times`;  
}, [count]); // Only re-run the effect if count changes
```





## Rules for Hooks

---

- Only call at Top Level (not inside loops, conditions, etc.)
- Only call from React functions **or** custom hooks (next slide)
- Many more pre-defined hooks:  
<https://reactjs.org/docs/hooks-reference.html>



## Custom hooks

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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

JS function name starts with use

**Must** follow Rules of Hooks (from previous slide)



## Using custom hooks

```
function FriendStatus(props) {  
  const isOnline = useFriendStatus(props.friend.id);  
  
  if (isOnline === null) {  
    return 'Loading...';  
  }  
  return isOnline ? 'Online' : 'Offline';  
}
```

```
function FriendListItem(props) {  
  const isOnline = useFriendStatus(props.friend.id);  
  
  return (  
    <li style={{ color: isOnline ? 'green' : 'black' }}>  
      {props.friend.name}  
    </li>  
  );  
}
```

Two different components  
using the same hook **do**  
**not** share state

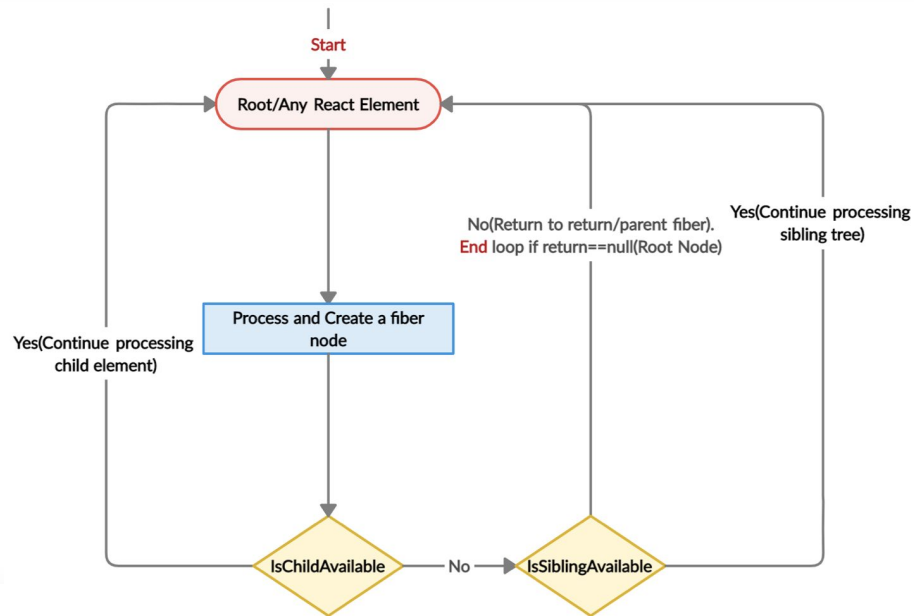
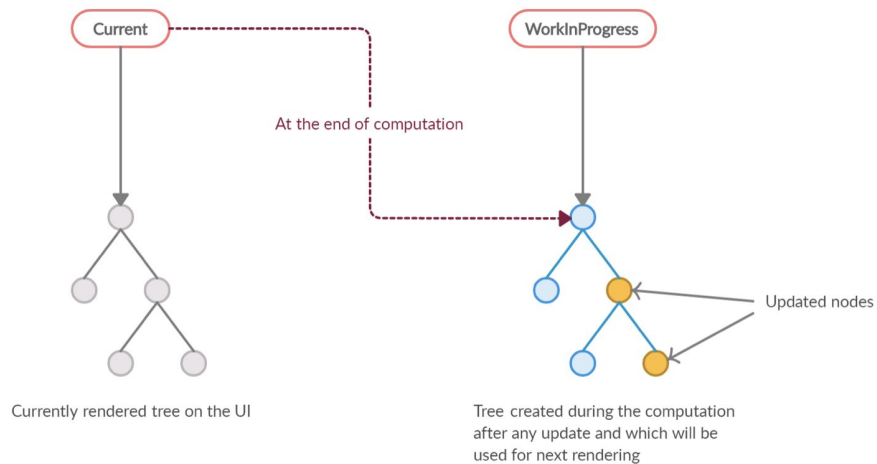


## How do Hooks work under the hood?

- ◉ In React, **reconciliation** and **render** are two different phases
- ◉ **Reconciliation** is the tree diffing algorithm in React used to say what changed (virtual DOM)
- ◉ **Render** uses that info to update the app view
- ◉ Reconciliation is implemented using **fibers**
  - a JavaScript object that contains information about a component, its input, and its output
  - a kind of virtual stack frame



# Fiber tree traversal





## Hooks are called in render

- Hooks use a “dispatcher” object
- When you call `useState`, `useEffect`, ... it passes the call to the dispatcher object
- The renderer is what executes the component function (based on the result of reconciliation)
- The renderer knows the context of the component and works through linked list of hooks



## More resources on React under the hood

---

- How does React actually work? (video)
  - <https://www.youtube.com/watch?v=7YhdqIR2Yzo>
- What is React Fiber?
  - <https://www.youtube.com/watch?v=OympFlwQFIw>
  - <https://blog.logrocket.com/deep-dive-react-fiber/>
- How do Hooks work?
  - <https://www.youtube.com/watch?v=1VVfMVOabx0>



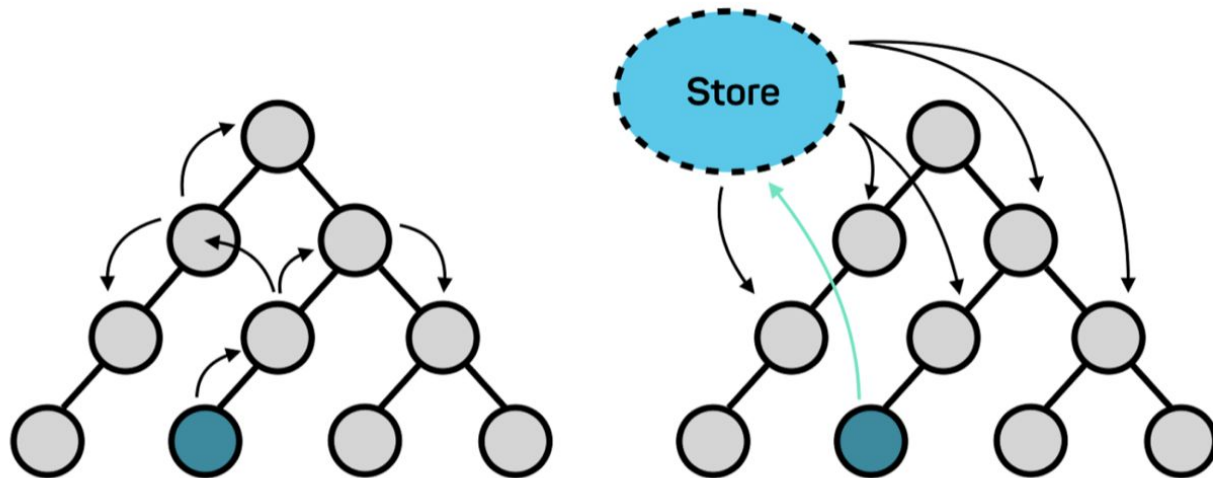
## State and props revisited

- ◉ In React, **state** is internal to components
- ◉ **props** are used to pass information from parents to children
- ◉ Children can pass information to parents, by passing a parent setter function as a prop in a child element
- ◉ But what about complex apps with shared state across components not in parent-child relationship?





## Global state



Component initiating change

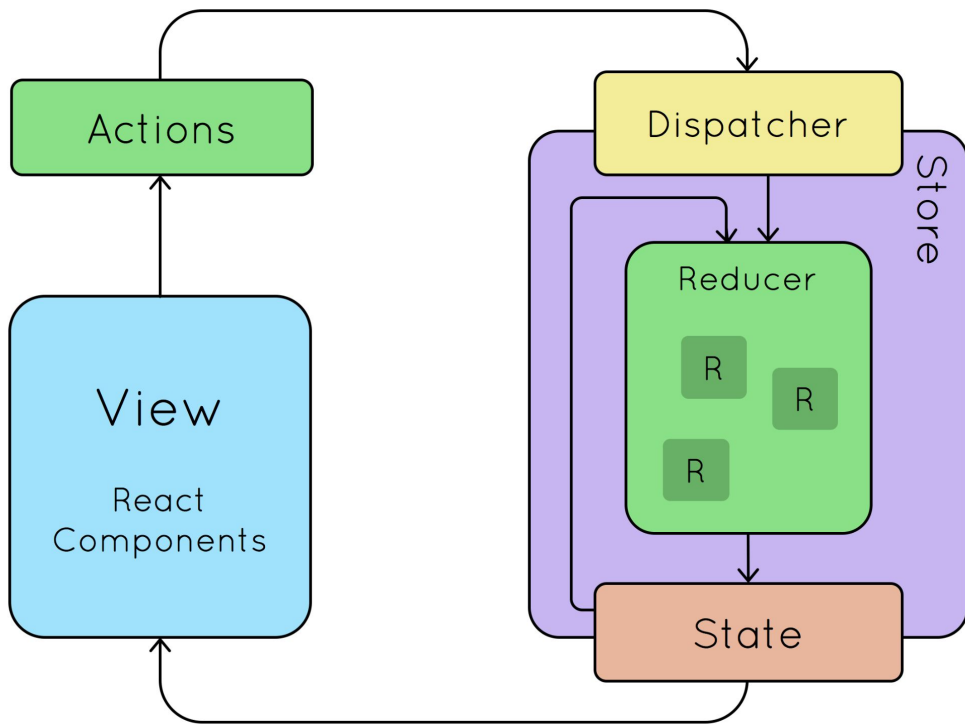


## Global state management

- Several libraries exist to manage global state
- Define actions to update the state, can be async functions (e.g., including a data fetch from server)
- Redux is the most popular
  - Verbose, lots of boilerplate
  - Best to use Redux-Toolkit for React >16.8 with function components and hooks
- We use Zustand in the labs
- `useContext` hook is useful for sharing global variables such as CSS themes, but can be inefficient for large apps (triggering updates of all components)



## Redux model

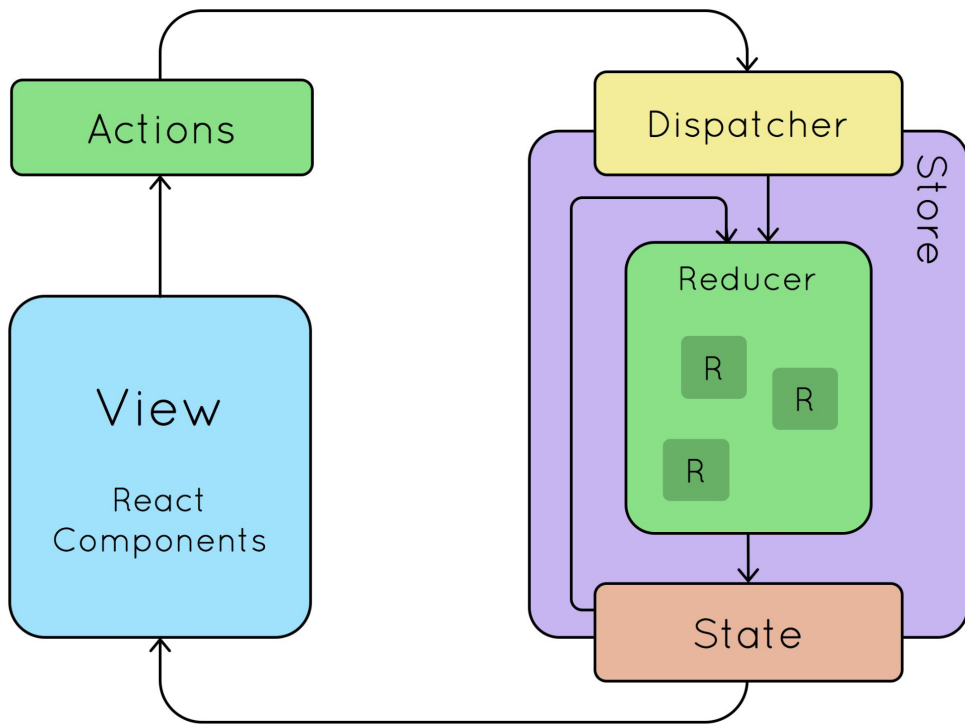


**Actions** = JavaScript objects that *describe* the action to be taken

```
//action to add a todo item
{ type: 'ADD_TODO', text: 'This is a new todo' }
//action that pass a login payload
{ type: 'LOGIN', payload: { username: 'foo', password: 'bar' }}
```



## Redux model

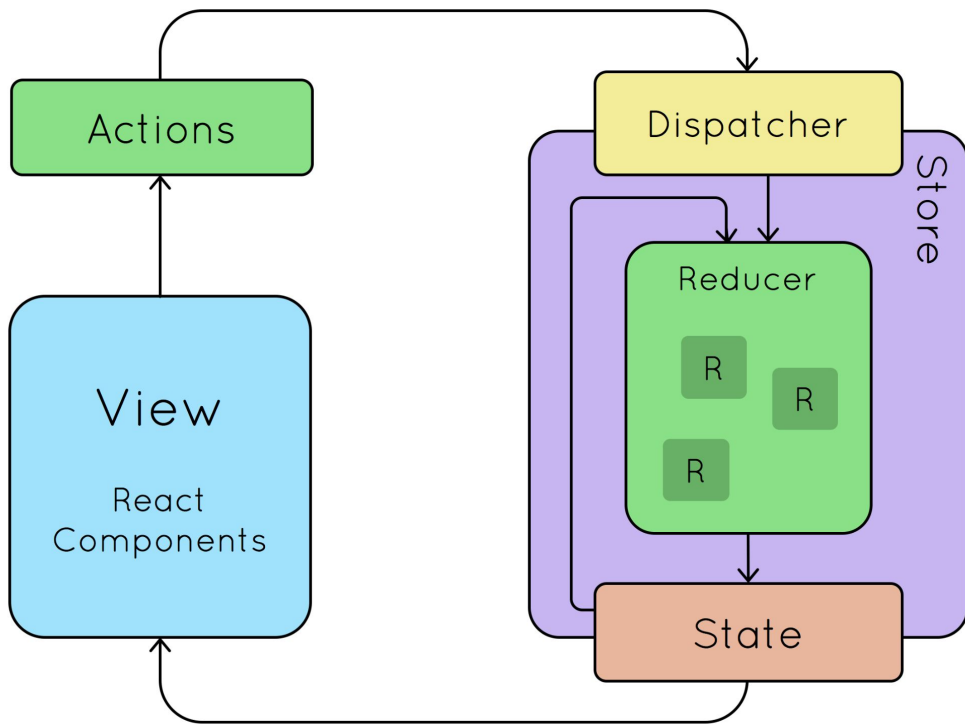


**Reducer** = *pure functions* that take an action and update the state

```
//takes in the current state and action
//updates the value based on the action's type
function counterReducer(state = { value: 0 }, action) {
  switch (action.type) {
    case 'INCREASE':
      return { value: state.value + 1 }
    case 'DECREASE':
      return { value: state.value - 1 }
    default:
      return state
  }
}
```



## Redux model



**Redux-toolkit**: slices combines all this to create less boilerplate code

```
import { createSlice } from '@reduxjs/toolkit'

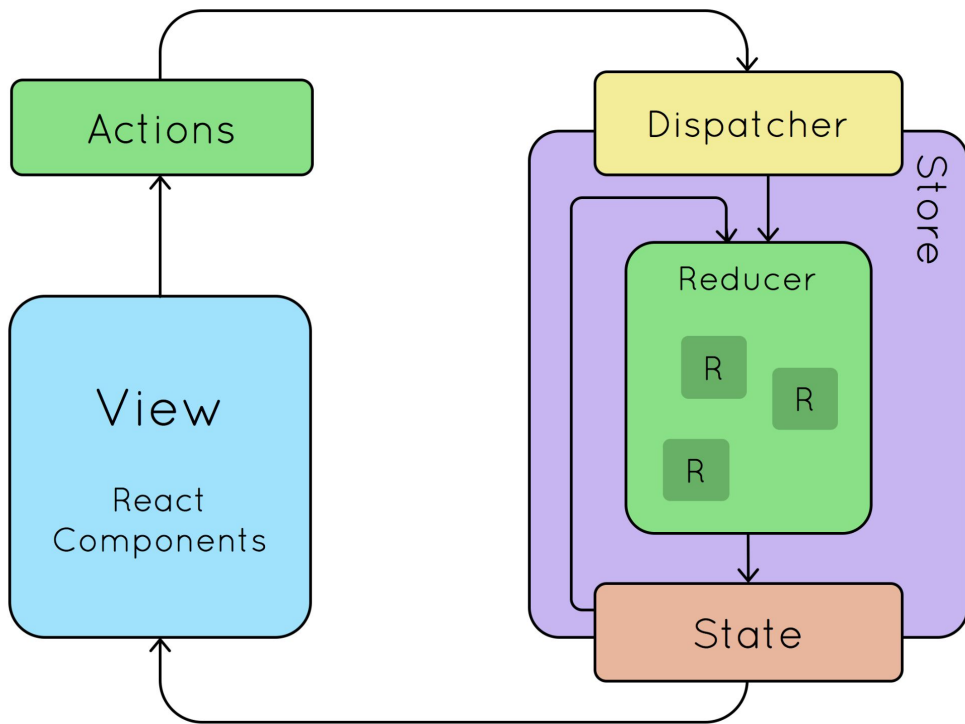
export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    increase: state => {
      state.value += 1
    },
    decrease: state => {
      state.value -= 1
    }
  }
})

// each case under reducers becomes an action
export const { increase, decrease } = counterSlice.actions

export default counterSlice.reducer
```



## Redux model



**Redux-toolkit**: `useSelector` and `useDispatch` hooks are used in the component

```
export function Counter() {  
  const count = useSelector(state => state.counter.value)  
  // in our slice, we provided the name property as 'counter'  
  // and the initialState with a 'value' property  
  // thus to read our data, we need useSelector to return the state.counter  
  
  const dispatch = useDispatch()  
  // gets the dispatch function to dispatch our actions  
  
  return (  
    <div>  
      <button onClick={() => dispatch(increase())}>  
        Increase  
      </button>  
      <p>{count}<p>  
      <button onClick={() => dispatch(decrease())}>  
        Decrease  
      </button>  
    </div>  
  )  
}
```



# Zustand

The screenshot shows a code editor with a dark theme. On the right side, there is a preview of a UI component. The code defines a Zustand state with a count of 1 and an increment function. The `Controls` component uses `useStore` to access the `inc` function and renders a button. The `Counter` component uses `useStore` to access the `count` and renders an `h1` element. The UI preview shows a dark blue box with a large white number '1' and a button labeled 'one up'.

```
import create from 'zustand'

const useStore = create(set => ({
  count: 1,
  inc: () => set(state => ({ count: state.count + 1 })),
}))

function Controls() {
  const inc = useStore(state => state.inc)
  return <button onClick={inc}>one up</button>
}

function Counter() {
  const count = useStore(state => state.count)
  return <h1>{count}</h1>
}
```



## useContext

1. Create a context object

```
const UserContext = createContext()
```





## useContext

1. Create a context object
2. Wrap components in context Provider (user is a state variable)

```
const UserContext = createContext()

<UserContext.Provider value={user}>
  <h1>{'Hello ${user}!'}</h1>
  <Component2 user={user} />
</UserContext.Provider>
```



## useContext

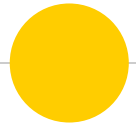
1. Create a context object
2. Wrap components in context Provider, and pass state variable (e.g. `user`) as value
3. In child component access the variable with `useContext`

```
const UserContext = createContext()
```

```
<UserContext.Provider value={user}>  
  <h1>{'Hello ${user}!'}</h1>  
  <Component2 user={user} />  
</UserContext.Provider>
```

```
const user = useContext(UserContext);
```

```
return (  
  <>  
    <h1>Component 5</h1>  
    <h2>{'Hello ${user} again!'}</h2>  
  </>  
)
```



# Routing



## Routing basics

- In a traditional non-SPA website, when you navigate to a new route (e.g. by clicking a link) it loads a new page from the server
- In SPA:
  - The page is not re-loaded
  - The link is intercepted and any new content needed from the server is made by client-side code
  - Do not need to re-render entire page, including framework JS (e.g. React), potentially saving data transfer/time.
  - Bonus: can update the page dynamically, e.g. using CSS transitions



## React Router basic concepts

---

**URL** – URL is address bar

**Location** – object based on window.location

**History** – object that subscribes to the browser's history stack

The history stack is changed via POP, PUSH, REPLACE operations

**Client-side routing** – programmatically changing the history stack without making a server request

**Route match** – when the URL matches a pattern a specific route is rendered



## Location object

pathname, search and hash  
come from the url

state is the current top of the  
history stack

key is a unique key (useful for  
client-side caching etc.)

```
{  
  pathname: "/bbq/pig-pickins",  
  search: "?campaign=instagram",  
  hash: "#menu",  
  state: null,  
  key: "aefz24ie"  
}
```



## Route config and matching

```
<Routes>
  <Route path="/" element={<App />}>
    <Route index element={<Home />} />
    <Route path="teams" element={<Teams />}>
      <Route path=":teamId" element={<Team />} />
      <Route path=":teamId/edit" element={<EditTeam />} />
      <Route path="new" element={<NewTeamForm />} />
      <Route index element={<LeagueStandings />} />
    </Route>
  </Route>
  <Route element={<PageLayout />}>
    <Route path="/privacy" element={<Privacy />} />
    <Route path="/tos" element={<Tos />} />
  </Route>
  <Route path="contact-us" element={<Contact />} />
</Routes>
```

- The route config is a tree of route elements.
- Matches can be **exact** or **dynamic** (e.g. `:teamId`)
- `path` prop defines the match
- `element` prop defines the component to be rendered



## Higher-order Components

```
class Welcome extends React.Component {
  render() {
    return (
      <div>Welcome {this.props.user}</div>
    );
  }
}

const withUser = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      if (this.props.user) {
        return (
          <WrappedComponent { ...this.props} />
        )
      }
      return <div>Welcome Guest!</div>
    }
  }
}

export default withUser(Welcome);
```

Wrapping components with  
props proxy





## Higher-order Components

Wrapping components with  
inheritance inversion

```
class Welcome extends React.Component {
  render() {
    return (
      <div>Welcome {this.props.user}</div>
    );
  }
}

const withUser = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      if (this.props.user) {
        return (
          <WrappedComponent { ...this.props} />
        )
      }
      return <div>Welcome Guest!</div>
    }
  }
}

const withLoader = (WrappedComponent) => {
  return class extends WrappedComponent {
    render() {
      const { isLoading } = this.props;
      if (!isLoading) {
        return <div>Loading...</div>;
      }
      return super.render();
    }
  }
}

export default withLoader(withUser(Welcome));
```



# Debugging React Apps

---



## Tips on debugging

---

- When you run in dev hot reloading is enabled
- Keep browser open with web page (e.g. `localhost:3000`) and console visible while developing
- Sometimes when it crashes (shows error message in browser) you need to reload the page



## console.log

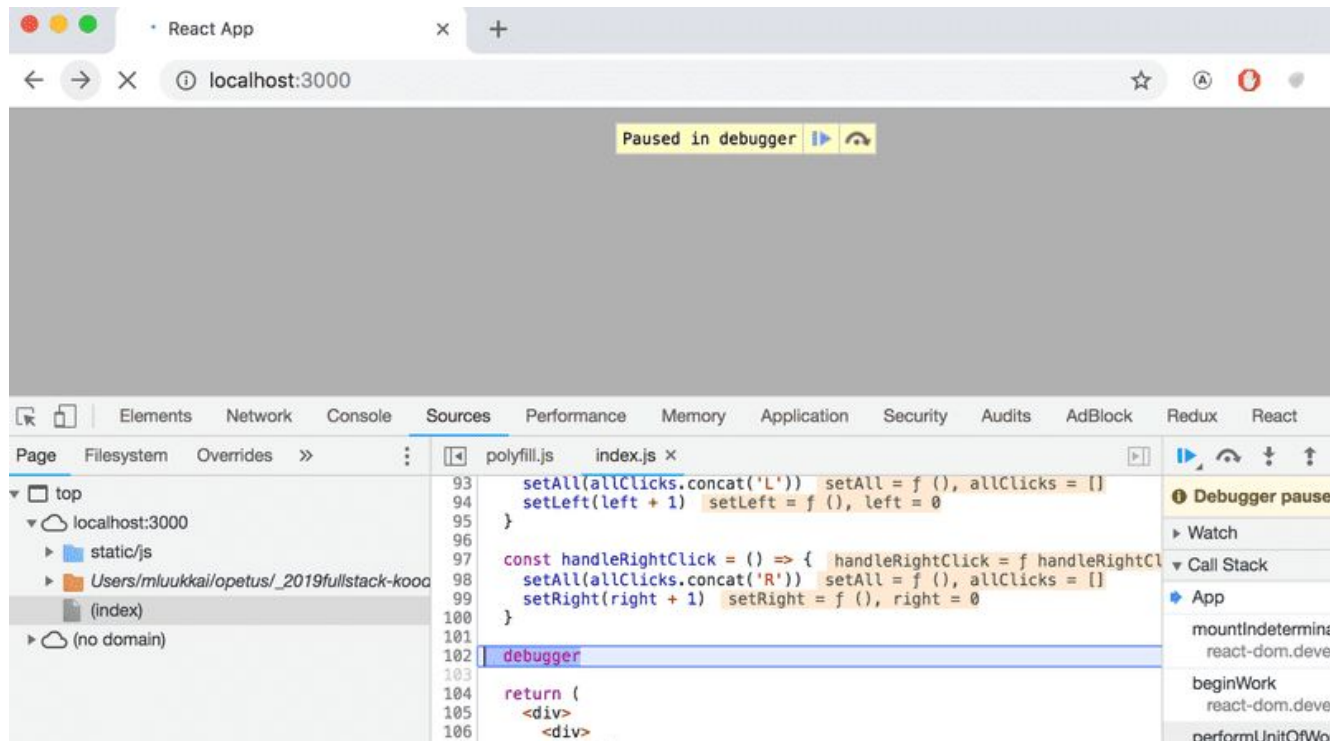
- `console.log` is still a great way to view the state of variables when components load
- Remember to remove these before production, though!

```
console.log('props value is', props)
```

Use comma, not + when concatenating complex objects to strings



## debugger in Chrome developer



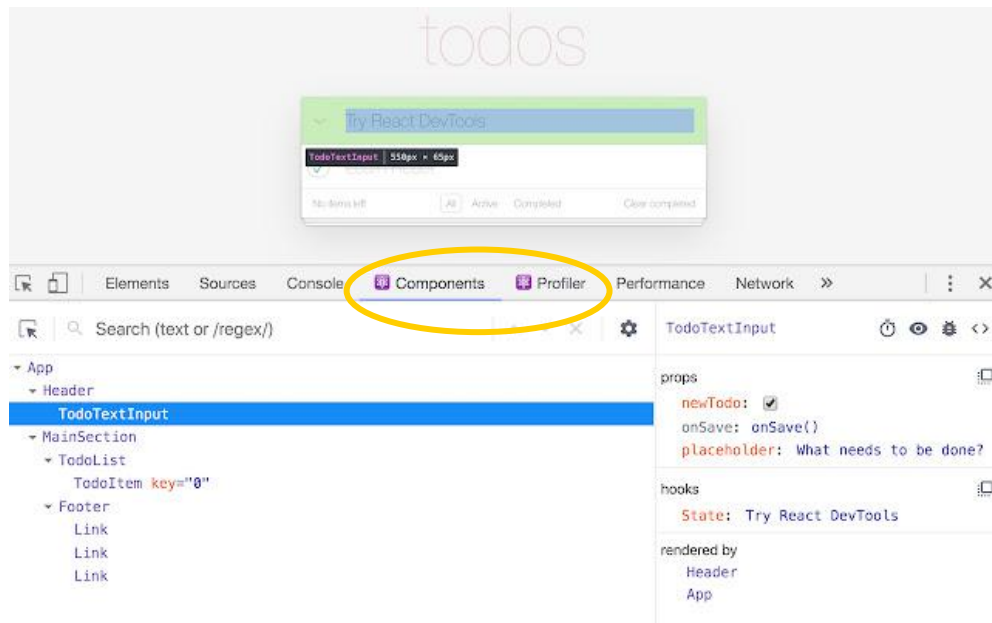
Can also set **breakpoints** manually.

Once it has stopped you can use the console to view the value of various variables.



## React developer tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>



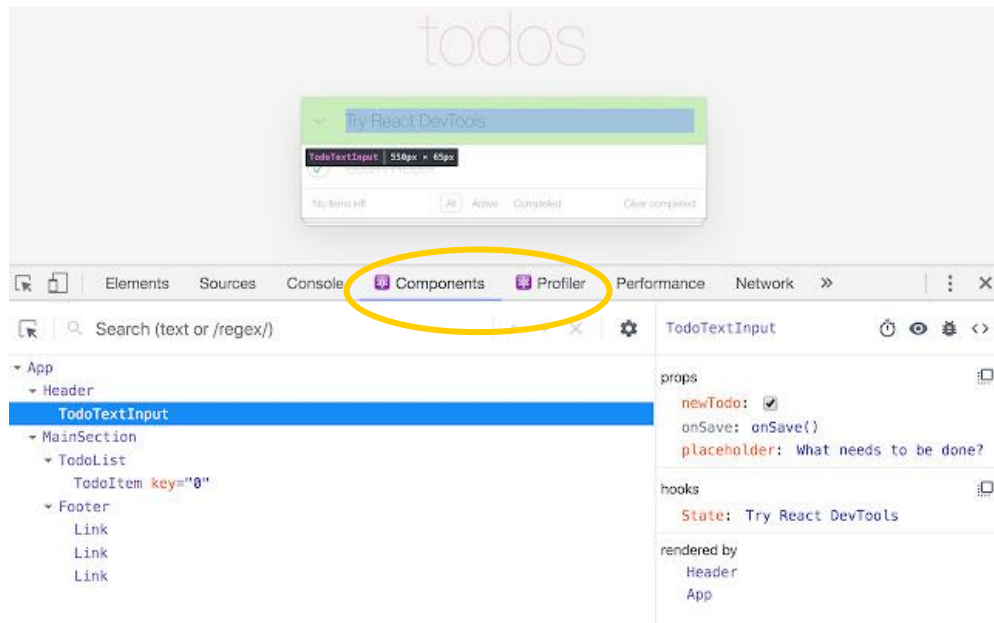
Adds a **Components tab** to the developer console

Can view component **state** and **props**, and **hooks** in order of definition



## React developer tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>



Adds a **Components** and **Profiler** tabs to the developer console

Can view component **state** and **props**, and **hooks** in order of definition

Profiler allows you to test performance of components