Yiyang Wang

https://utoronto.zoom.us/j/2210147631
(Passcode: 123456)

CSC309 Week 8

React Practice & Hooks & Server Actions

HTTPS://WWW.CS.TORONTO.EDU/~KIANOOSH/COURSES/CSC309H5/

5 March, 2025

run demo

In your terminal :

git clone https://github.com/yiyangww/tut-309.git

cd tut-309

npm install

npm run dev

Today’s Lab

many codes demo … (don’t worry, they are simple!)

React and Hooks

custom hooks

server actions

React Quick Review and More…

- declarative approach: You define the UI, React updates it
- component-based: root component → App.jsx
- core concepts:

  component: building blocks, functional component / class component

  (capitalized naming convention)

React Quick Review and More…

- declarative approach: You define the UI, React updates it
- component-based: root component → App.jsx
- core concepts:

  component: building blocks, functional component / class component

  (capitalized naming convention)

  props: properties passed from a parent component to a child component

  state: app’s dynamic data managed within a component

- one-way data flow: clear state management

JSX

- Next.js uses JSX just like React. (.js, .jsx, .ts, or .tsx)
- Benefits :

  → Enhanced Readability: Clearly represents the UI structure within the code, making it more intuitive.

  → Seamless JavaScript Integration: Embeds dynamic data and expressions effortlessly.

  → Component-Centric Approach: Simplifies the creation and reuse of modular components

dynamic logic

```
const isLoggedIn = true;
const message = <h1>{isLoggedIn ? "Welcome" : "Please Log In"}
</h1>;
```

event handler

```
function handleClick() {
  alert("Button clicked!");
}
const button = <button onClick={handleClick}>Click
Here</button>;
```

Destructure Objects

```
const user = { firstName: "Cool", lastName: "Web", isLoggedIn: true
const greeting = (
  <h1>
    {user.isLoggedI
      ? `Welcome, ${user.firstName} ${user.lastName}`
      : "Please log in."}
  </h1>
);
```

Inline Styling

```
const style = { color: "blue", fontSize: "20px"
const title = <h1 style={style}>Styled Text</h1>;
```

spread attributes

```
const props = { className: "btn", type: "submit", disabled: true
const button = <button {...props}>Submit</button>;
```

arrays and lists

```
const items = ["React", "JSX", "Components"];
const list = (
  <ul>
    {items.map((item) => (
      <li key={item}>{item}</li>
    ))}
  </ul>
);
```

Traditional DOM vs New DOM

old: manually find elements, more error-prone, non-reusable

an old DOM example

```
<div>
  <p id="count">Count: 0</p>
  <button id="increment">Increment</button>
</div>
<script>
  let count = 0;
  document.getElementById("increment").addEventListener("click", () => {
    count++;
    document.getElementById("count").textContent = `Current Count:
${count}`;
  });
</script>
```

- virtual DOM: React updates(automatically) changed DOM parts

```
"use client";
import { useState } from "react";

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

  const increment = () => setCount((prevCount) => prevCount +
1);
  return (
    <div>
      <p>Simple Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;
```

Lift State Up Example (here I wrote with react vite)

Before I demo, a short explanation about …

main.jsx: starting point ; initializes react; loads App.jsx (strict mode?)

App.jsx : as a container; defines the core UI; renders child components

demo

```jsx
function App() {
  const [count, setCount] = useState(0);

  const increment = () => setCount(prevCount => prevCount +
1);

  return (
    <>
      <Counter count={count} onClick={increment} />
      <Counter count={count} onClick={increment} />
    </>
  );
}

export default App;
```

parent

child

```jsx
function Counter(props) {
  return (
    <div>
      <p>Current Count: {props.count}</p>
      <button onClick={props.onClick}>Increment</button>
    </div>
  );
}

export default Counter;
```

props.count is dynamically inserted into the JSX; parent pass count to child

Will this code work?

```
function Counter() {
  let count = 0;

  const increment = () => {
    count += 1;
  };

  return (
    <div>
      <p>Current Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

Will this code work?

```
function Counter() {
  let count = 0;

  const increment = () => {
    count += 1;
  };

  return (
    <div>
      <p>Current Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

No! It won’t update in UI!

→ sol: useState

Built In Hooks

hook name → demo component

- useState → Counter
- useReducer → NewCounter
- useEffect → Timer
- useContext (mentioned in week 7 lab)

more:

https://react.dev/reference/react/hooks

useState vs useReducer (demo)

initial state

better for less complex data

hard to maintain when state gets more complex

initial state + reducer function

better for more complex data

requires more prepwork

useEffect (demo)

componentDidMount, componentDidUpdate, componentWillUnmount

useEffect replace these lifecycle methods!

dependency array: empty/state variables/props(from parent)

```
useEffect(() => {
  // Code runs when the component mounts or when dependencies change
  return () => {
    // Cleanup code runs before the component unmounts
    // OR before the effect re-runs due to dependency changes
  };
}, [dependencies]);
```

side effects

useEffect is primarily used in React to handle side effects, which are operations that do not directly affect UI rendering but are still necessary.

Common side effects include:

Fetching data (API requests) – Loading data from a server when a component mounts.

Subscribing to events – Listening to browser events, WebSocket connections, etc.

Manipulating the DOM – Updating document.title, setting focus on an element, etc.

Cleanup tasks – Removing event listeners, clearing timers, or canceling API requests when a component unmounts.

Built In Hooks Cont’d

useState: Manages state within functional components.

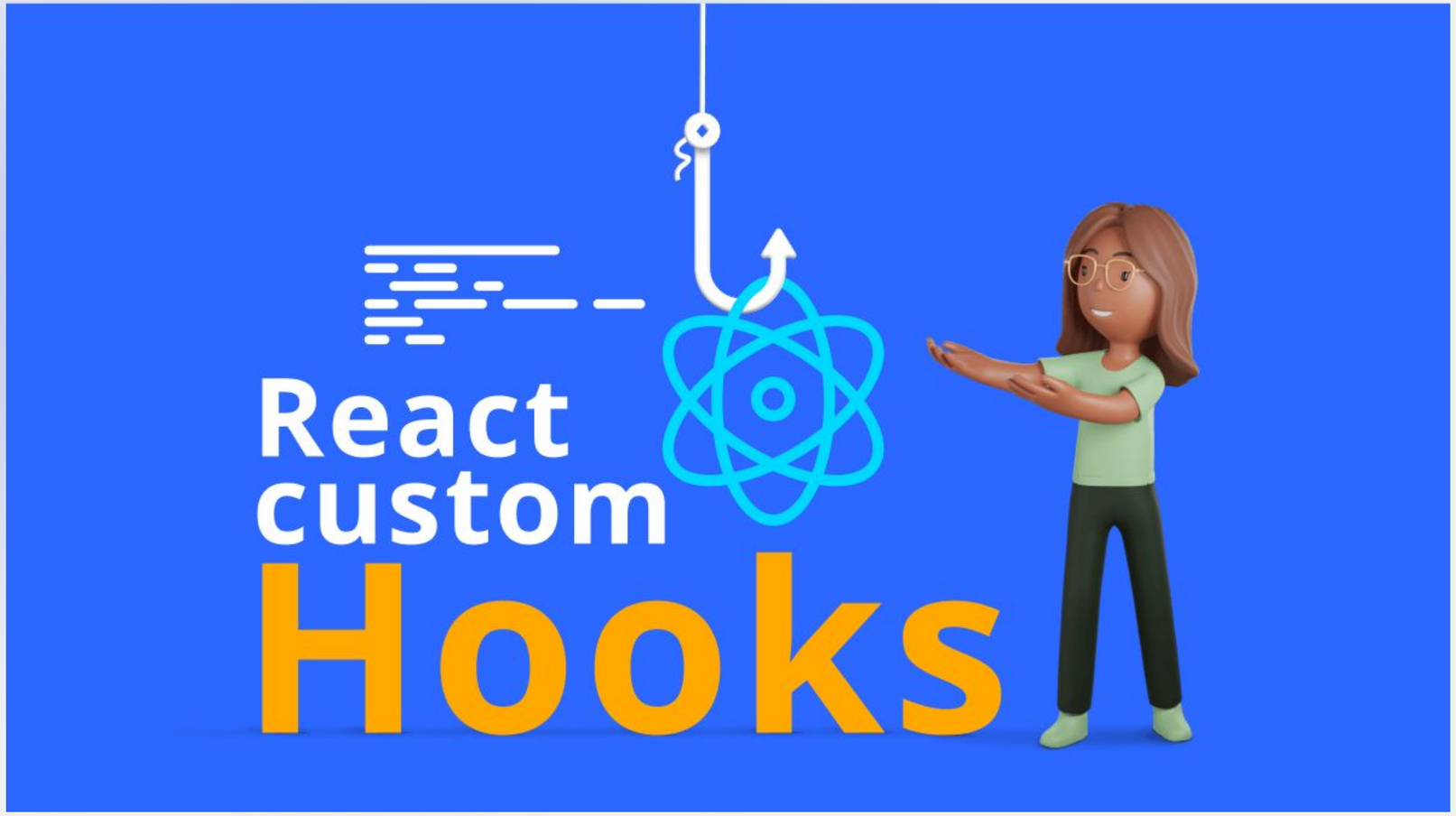
useReducer: Manages complex state logic with actions and reducers.

useEffect: Handles side effects like data fetching and subscriptions.

useContext: Provides global state sharing without prop drilling.

Custom Hooks

Tidy up ur code!



Code Reusability

Cleaner and More Readable Code

Separation of Concerns

Improves Testability

Custom Hooks (demo)

src/hooks/useConsoleLog.js

```
import { useEffect } from "react";

export default function useConsoleLog(value) {
  useEffect(() => {
    console.log(value);
  }, [value]);
}
```

src/components/ConsoleLog.js

```
"use client";
import { useState } from "react";
import useConsoleLog from "@/hooks/useConsoleLog";

export default function ConsoleLog() {
  const [count, setCount] = useState(0);
  useConsoleLog(count);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Plus
1</button>
    </div>
  );
}
```

Custom Hooks

src/hooks/useSwitch.js

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

export default function useSwitch(initialValue) {
  const [state, setState] = useState(initialValue);
  const boolSwitch = () => setState((prev) =>
!prev);
  return [state, boolSwitch];
}
```

src/components/SwitchButton.js

```
"use client";
import useSwitch from "@/hooks/useSwitch";

export default function SwitchButton({ initialValue = false })
{ const [isOn, switchIsOn] = useSwitch(initialValue);

  return (
    <div>
      <p>The button is: {isOn ? "ON" : "OFF"}</p>
      <button onClick={switchIsOn}>Switch</button>
    </div>
  );
}
```

Sever Actions

1. Server-side execution

   Functions marked with "use server" only run on the server and are not exposed to the client.

2. No need for separate API routes

   Unlike traditional pages/api or app/api, you don’t need to manually create API endpoints.

3. Improved security

   The client cannot see the implementation of the function, reducing exposure to security risks.

4. Simplified data mutations

   Server Actions allow you to modify the database, update session data, or interact with external APIs without requiring additional client-side fetch calls.

A very simple example (demo)

src/app/server-actions/page.js

```
"use client";
import { useRouter } from "next/navigation";
import { useState } from "react";
import { greetUser } from "../../actions";

export default function ServerActionsDemo() {
  const router = useRouter();
  const [name, setName] = useState("");
  const [message, setMessage] = useState("");

  async function handleSubmit(e) {
    e.preventDefault();
    const response = await greetUser(name);
    setMessage(response);
  }
```

```
  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Server Actions Demo</h1>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          placeholder="Enter your name"
          value={name}
          onChange={(e) => setName(e.target.value)}
          required
        />
        <button type="submit">Submit</button>
      </form>
      {message && <p>{message}</p>}
      <br />
      <br />
      <button onClick={() => router.push("/")}>Return to Home</button>
    </div>
  );
}
```

src/actions.js

```
"use server";

export async function greetUser(name) {
  return `Hello, ${name}! Welcome to Server
Actions`;
}
```

modify action.js

```
"use server";
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

export async function greetUser(name) {
  // store to db
  const user = await prisma.user.create({
    data: { name },
  });

  return `Hi, ${user.name}! Your name is stored in
database!`;
}
```

Counter in Server Actions

demo

some limitations

- No File Upload Support

Server Actions cannot handle file uploads directly.

You still need an API Route (pages/api) or an external service.

- No Direct Support for JWT Authentication

Server Actions do not automatically handle JWT authentication.

You'll need API Routes or middleware to verify tokens.

- No Middleware or Interceptors

Unlike API Routes, Server Actions do not support Next.js Middleware (middleware.ts).

You cannot intercept or modify requests globally.

00:30

1. Is this a valid useState hook invocation and destructuring?

```
const [car, setCar] = useState({ brand: 'BMW', mileage: 0})
```

(Single choice)

- ◯ A. Yes
- ◯ B. No
- ◯ C. It would be valid, if it was spread over multiple lines.

01:00

2. Consider the following code:

```
const [person, setPerson] = useState({ name: 'John', age: 21})
```

Imagine you're using a setPerson() state-updating function to update the value of the state variable named person. You only want to update the value of age, from 21 to 22. Choose the correct code snippet to do that. (Single choice)

- ◯ A. `setPerson(prev => ({ ...prev, age: 22 }));`
- ◯ B. `setPerson(() => ({ age: 22 }));`
- ◯ C. `setPerson(person.age = 22);`

00:30

3. What is the name of the second argument of the useEffect() call? (Single choice)

- A. the dependencies object
- B. the dependency array
- C. the destructured object
- D. the callback function

01:00

4. If I need to navigate to another page, which of the following methods satisfy the Single Page Application (SPA) principle? (Multiple choice)

- [ ] A. Using router.push() in Next.js <button onClick={() => router.push("/")}>Return to Home</button>
- [ ] B. Using a regular <a> tag inside a button Return to Home
- [ ] C. Using useNavigate() in React Router <button onClick={() => navigate("/")}>Return to Home</button>
- [ ] D. Using Link with a button in Next.js <Link href="/">Return to Home</Link>

Thank You!