Next Gen Engineering
School of Tech - INDIA

React.js 3

Lifecycle & Side Effects

Overview

- The Component Lifecycle
- React Hook: useEffect
- Fetching Data
- Testing user events

Objectives

- Understand React's component lifecycle
- Have working knowledge of `useEffect`
- Know how to perform side-effects such as fetching data
- Understand how to simulate user events and make assertions on them

The React sessions

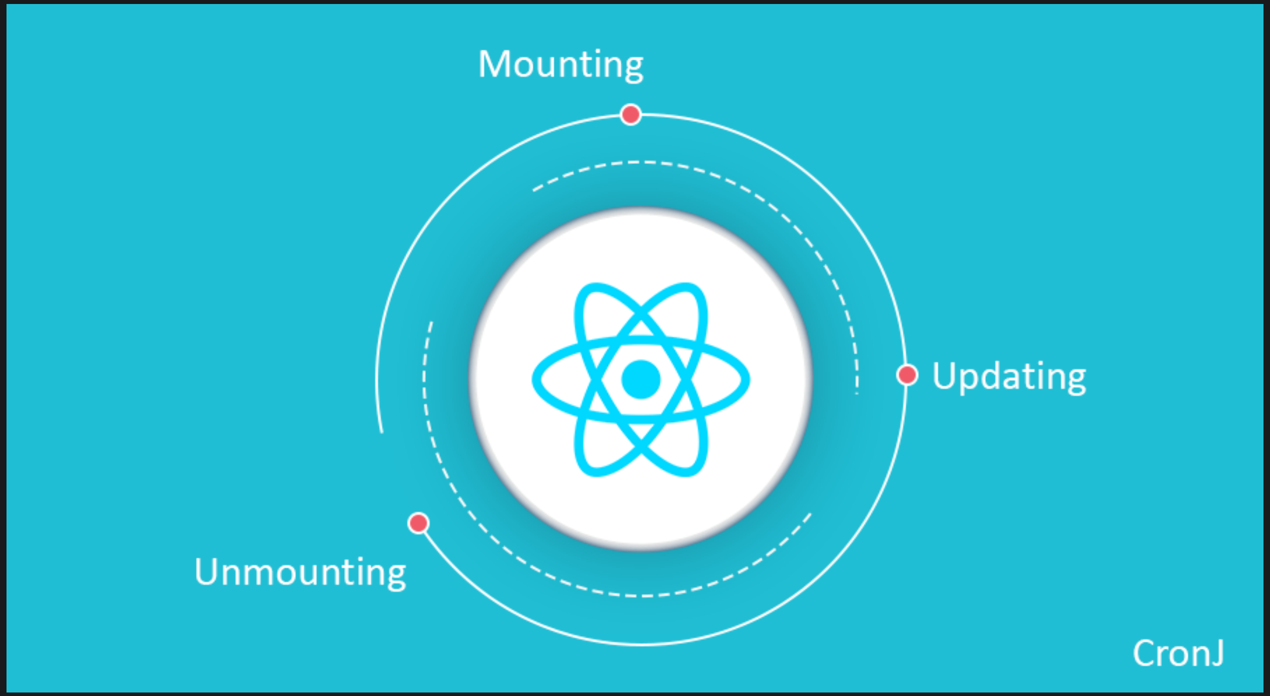
We have 4 React sessions:

- Components: Introduction to React; Components & JSX; Testing Components ✅
- State: Virtual DOM; Data Flow; State Management with the `useState` React Hook ✅
- Side Effects: Component Lifecycle; Fetching Data with the `useEffect` React Hook; Testing User Events (this one)
- Routing: React Router; Building & Deployment

Component Lifecycle

Mount -> Update -> Unmount
React components have a lifecycle which we can hook into and trigger side-effects, such as:

- Asynchronously interacting with an API / Database
- Modifying state or performing clean-up functions



Component Lifecycle

React Hook: useEffect

`useEffect` allows us to perform side-effects by hooking into a component's lifecycle events.
`useEffect` accepts two parameters: A function to execute (inline or referenced), and an optional dependency array.
The second parameter describes where in the lifecycle the side-effect should be performed. Examples on the next slide.
Codepen [React Lifecycle]
useEffect

```
// Run when the component mounts
useEffect(someFunction, [])

// Run on every update (including mount)
useEffect(someFunction)

// Run both on mount and when variable1 changes
useEffect(someFunction, [variable1])

// Function to run
const someFunction = () => {
  // Code here gets run on mount & updates
  console.log("mounted")
  // Returned function is run on unmount
  return () => {
    console.log("unmounted")
  }
}
```

Clean up

Example is in `./examples/use-effect-cleanup`

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('time is up')
  }, 1000)

  return () => {
    clearInterval(timer)
  }
}, [])
```

Emoji Check:

Do you understand how to use the `useEffect` hook?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Gotcha

Watch out for side-effects that depend on a variable in state while also setting that variable. This will create an undesirable infinite update loop.
Codepen [Infinite Loop]

Fetching Data

In this example we make use of the `useEffect` hook to fetch some data when the app starts (mount).
Notice that `fetch` requests are separated from the component and instead defined as their own pieces of discrete logic. This is good practice and conforms to "separation of concerns", and "reusability" principles.
Codepen [Fetching Data]

Exercise - Country App - 30mins

Instructor to distribute exercise:

> See `./exercises/react-countries-part-1/README.md`

Emoji Check:

How do you feel about the first part of the exercise?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Next Gen Engineering
School of Tech - INDIA

Testing - Simulating user-events

We have a component like this, that the user can click on:

```
type CountrySelectProps = {
    options: Array<string>
    onSelect: (item: string) => void
    selected: string
}

const CountrySelect = ({ options, onSelect, selected }: Countr
  <select onChange={(event) => onSelect(event.target.value)}
    value={selected}>
    {options.map((item) => (
      <option key={item} value={item}>
        {item}
      </option>
    ))}
  </select>
)
```

On the next slides we'll build up a test for how it renders and what happens when it is clicked.

What to test?

We want to test:

- All the options get added to the select box
- When I select an option, the `onSelect` handler is called with the correct value
- The default value is pre-selected in the select box (in the exercises later)

Select box selector test

```
it("should render the countries", () => {
  const countries = ["Japan", "Italy"]
  render(<CountrySelect options={countries} />)

  const selectElem = screen.getByRole("combobox")
  expect(selectElem).toContainElement(screen.getByText("Japan"
  expect(selectElem).toContainElement(screen.getByText("Italy"
})
```

Here we used `getByRole` to find the select element - `combobox` is an old name.

Testing an event handler

In the `CountrySelect` component we have a callback function for the `onSelect` event.
We can test that too.

The handler test

We can start with the test method and the Arranging:

```
it("should trigger the handler correctly", () => {
  // Arrange
  const countries = ["Japan", "Italy"]
  const mockHandler = jest.fn()
  // Act ...
  // Assert ...
})
```

Here we make a blank `mock` function to use for a dummy callback handler.

Act 1 - initial render

Then we can use the `CountrySelect` component to render it:

```
// Arrange
render(<CountrySelect options={countries} onSelect={mockHandle
```

Act 2 - trigger handler

To do this we need a new import (and in `package.json` of course!):

```
import userEvent from "@testing-library/user-event"
```

And then:

```
// Act - trigger handler
const selectElem = screen.getByRole("combobox")
const optionElem = screen.getByText("Japan")
userEvent.selectOptions(selectElem, optionElem)
// Assert
expect(mockHandler).toBeCalledWith("Japan")
```

This is the html Select element version, there are others.

userEvents

These are the various fake events provided by the test library:

- `click(element)`
- `dblClick(element)`
- `type(element, text)`
- `upload(element, file)`
- `clear(element)`
- `selectOptions(element, values)`
- `deselectOptions(element, values)`
- `tab()`
- `hover(element)`
- `unhover(element)`
- `paste(element, text)`

Testing - Best Practices

- Small components are far easier to test than larger ones
- Don't obsess over test coverage, 100% coverage is not necessary
- Focus on testing logical operations and control flow

Emoji Check:

Do you understand how to test with simulated user events?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Exercise - Country App continued - 30mins

Instructor to distribute exercise:

> See `./exercises/react-countries-part-2/README.md`

Overview - recap

- The Component Lifecycle
- React Hook: useEffect
- Fetching Data
- Testing user events

Objectives - recap

- Understand React's component lifecycle
- Have working knowledge of `useEffect`
- Know how to perform side-effects such as fetching data
- Understand how to simulate user events and make assertions on them

Emoji Check:

On a high level, do you think you understand the main concepts of this session? Say so if not!

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively