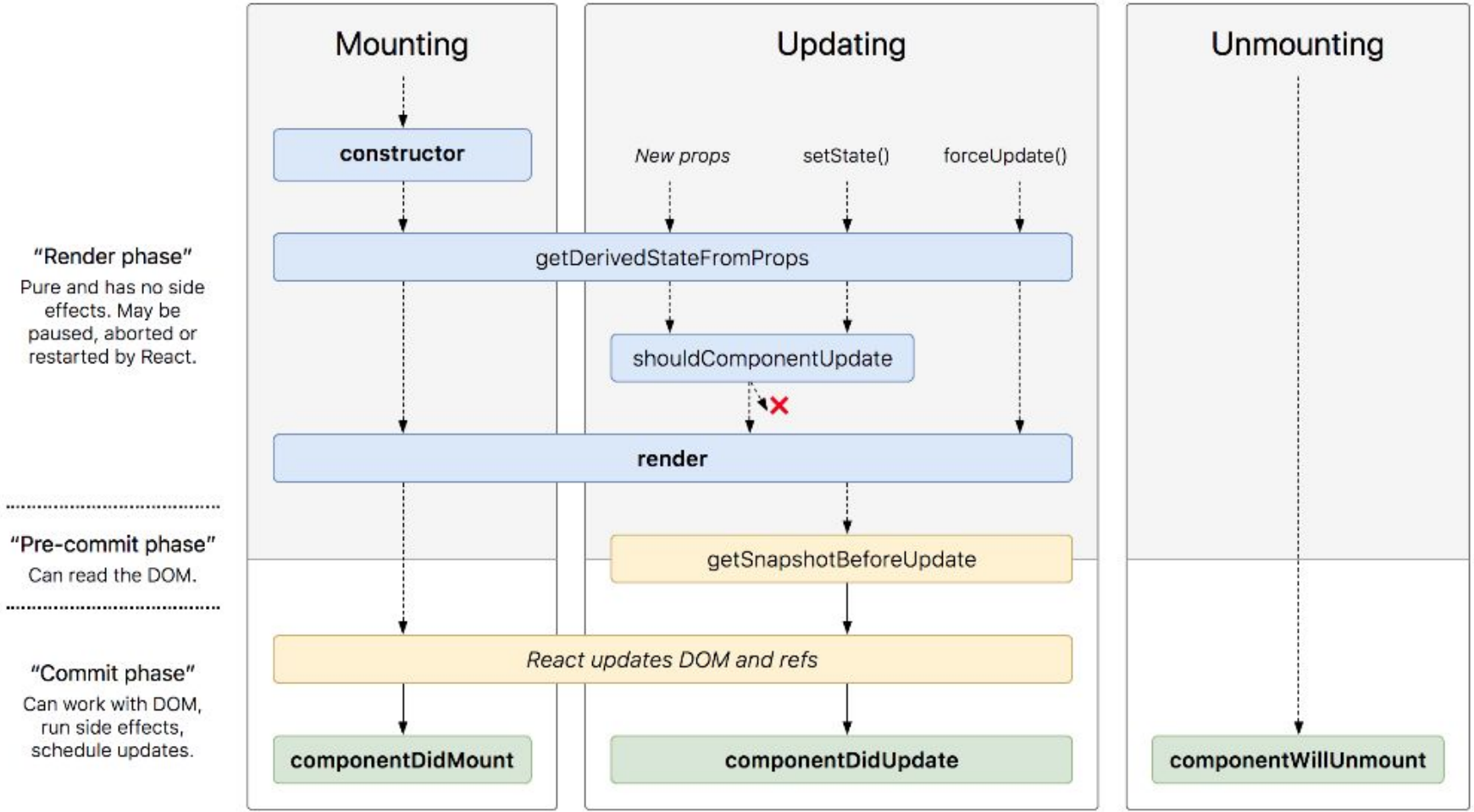
Lifecycles and Hooks

React 16.8 released

LifeCycles

Lifecycles

Durante la vida de un componente de React, se ejecutan varios métodos, en función del momento. A estos métodos se les llama métodos del ciclo de vida.


Mounting
Updating
Unmounting
constructor
New props
setState()
forceUpdate()
getDerivedStateFromProps
shouldComponentUpdate
render
getSnapshotBeforeUpdate
React updates DOM and refs
componentDidMount
componentDidUpdate
componentWillUnmount
"Render phase"
Pure and has no side effects. May be paused, aborted or restarted by React.
"Pre-commit phase"
Can read the DOM.
"Commit phase"
Can work with DOM, run side effects, schedule updates.

Ciclos de vida obsoletos

ComponentWillMount(): Por problemas de sincronía, se recomienda llevar el código al constructor o usar **componentDidMount()**;

Ciclos de vida obsoletos

---

componentWillReceiveProps(): Por problemas de renderizado y sincronía recomiendan utilizar **componentDidUpdate()** y **getDerivedStateFromProps()**

Ciclos de vida obsoletos

---

componentWillUpdate(): Se invoca antes del renderizado cuando llegan nuevas props o nuevo estado, no se puede hacer this.setState({}). Puede ser remplazado por **componentDidUpdate()** y **getSnapshotBeforeUpdate()** que se utiliza sobretodo para capturar cierta información del DOM como por ejemplo la posición del scroll.

Hooks

```
class ButtonCounter extends React.Component {
  constructor() {
    super()
    this.state = { clicks: 0 }
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    this.setState({ clicks: ++this.state.clicks })
  }

  render() {
    return (
      <Button
        onClick={this.handleClick}
        text={`You've clicked me ${this.state.clicks} times!`}
      />
    )
  }
}
```

¿Qué son los hooks?

Los **hooks** son una nueva API de la librería de React que nos permite tener estado, y el ciclo de vida en los componentes creados con una **function.**

Hasta ahora para poder acceder a todas las posibilidades de la librería creabamos componentes con **class.**

Hook significa gancho y, precisamente, lo que hacen, es que te permiten enganchar tus componentes creados con **function** a todas las características que ofrece React.

Sintaxis de un componente creado con function

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

function Contador() {

  const [count, setCount] = useState(0)

  return (
    <div>
      <p>Has hecho click {count} veces</p>
      <button onClick={() => setCount(count + 1)}>
        Haz click!
      </button>
    </div>
  )
}
```

Hooks básicos

useState

Con el método useState declaramos el estado del componente, nos devuelve un **array** que tendrá el valor del estado y un **método** para actualizarlo.

```
const [ count, setCount] = useState(0);
```

Con *array destructuring* extraemos la primera posición del array, donde tendremos el valor del state, al que hemos llamado count. En la segunda posición, a la cual hemos llamado setCount nos devuelve el método que ejecutaremos para actualizar el estado.

```
setCount(count + 1)
```

useState

```jsx
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>
  );
}
```

useEffect

Hook que recibe como parámetro una función que se ejecutará cada vez que nuestro componente se renderice, ya sea por un cambio de estado, por recibir props nuevas o, y esto es importante, porque es la primera vez que se monta.

```
useEffect(()=> {

 console.log(“render!”);

})
```

useEffect

Cuando se cambia el estado:

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

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

  useEffect(() => {
    // Actualiza el title de la página en cada click!
    document.title = `Has hecho clic ${count} veces`
  })

  return (
    <div>
      <span>El contador está a {count}</span>
      <button onClick={() => setCount(count + 1)}>
        Incrementar contador
      </button>
    </div>
  )
}
```

useEffect

Cuando necesitamos

escuchar eventos:

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

function ShowWindowWidth() {
  const [width, setWidth] = useState(0);

  useEffect(() => {
    // Creamos una función para actualizar el estado con el clientWidth
    const updateWidth = () => {
      const width = document.body.clientWidth
      console.log(`updateWidth con ${width}`)
      setWidth(width)
    }
    // Actualizaremos el width al montar el componente
    updateWidth()
    // Nos suscribimos al evento resize de window
    window.addEventListener("resize", updateWidth)

    // Devolvemos una función para anular la suscripción al evento
    return () => {
      window.removeEventListener("resize", updateWidth)
    }
  })

  return (
    <div>
      <span>Width es de {width}px</span>
    </div>
  )
}
```

useEffect

Cuando hay cambio de props:

```
function PokemonInfo({ name = "pikachu" }) {
  const [pokemonInfo, setPokemonInfo] = useState(null);

  useEffect(
    () => {
      fetch(`https://pokeapi.co/api/v2/pokemon/${name}`)
        .then(res => res.json())
        .then(pikachu => {
          console.log(pikachu);
          setPokemonInfo(pikachu);
        });
    },
    [name]
  );

  return (
    pokemonInfo && (
      <span>
        La pokeId es #{pokemonInfo.id} y su nombre es {pokemonInfo.name}
      </span>
    )
  );
}
```

useEffect: cómo evitar que se vuelva a ejecutar

Por defecto los efectos se disparan cada vez que se realiza un nuevo renderizado pero podemos evitar que el efecto se vuelva a ejecutar pasándole un segundo parámetro al hook.

Con éste parámetro evitamos crear bucles infinitos o simplemente no queremos que nuestro componente se renderice cada vez que haya un cambio de estado o por nuevas props.

El parámetro es un array con todos los valores de los que depende nuestro efecto, de forma que sólo se ejecutará cuando ese valor cambie.

Si dejamos el array vacío React entenderá que no depende de ningún valor, por lo tanto solo se renderizara una vez y también se desmontará.

Reglas de los hooks

Llama Hooks solo en el nivel superior

No llames Hooks dentro de ciclos, condicionales, o funciones anidadas.

En vez de eso, usa siempre Hooks en el nivel superior de tu función en React. Siguiendo esta regla, te aseguras de que los hooks se llamen en el mismo orden cada vez que un componente se renderiza. **Esto es lo que permite a React preservar correctamente el estado de los hooks entre múltiples llamados a useState y useEffect.**

Reglas de los hooks

Llama Hooks solo en funciones de React

No llames Hooks desde funciones JavaScript regulares. En vez de eso, puedes:

- Llama Hooks desde componentes funcionales de React.
- Llama Hooks desde Hooks creados manualmente.

Siguiendo esta regla, te aseguras de que toda la lógica del estado de un componente sea claramente visible desde tu código fuente.

Beneficios de utilizar hooks

Los hooks surgen para:

- Compartir y reutilizar lógica entre componentes
- Reducir el tamaño de componentes muy complejos
- Eliminar la confusión que generan las clases.

Usando hooks conseguimos código más limpio y ordenado, también se evita el uso de *this* y sus complejidades, las funciones son procesadas más rápido por la máquina que las clases, en definitiva el resultado es una aplicación más eficiente y rápida.


DEVELOPER
CLASS COMPONENTS
REACT HOOKS
imgflip.com

Enlaces de referencia

Sobre Hooks:

https://midu.dev/react-hooks-introduccion-saca-todo-el-potencial-sin-class/

https://es.reactjs.org/docs/hooks-intro.html

Más info sobre los ciclos de vida:

https://blog.bitsrc.io/understanding-react-v16-4-new-component-lifecycle-methods-fa7b224efd7d