



Clase 11. React JS

RENDERIZADO CONDICIONAL



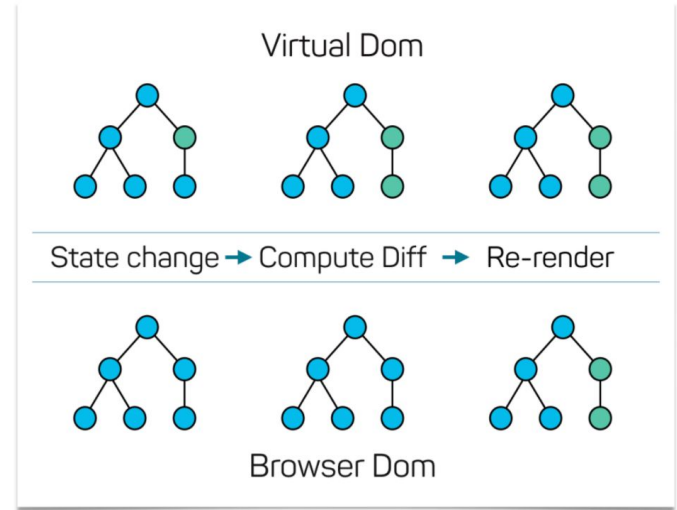
OBJETIVOS DE LA CLASE

- Profundizar sobre renderizado condicional y sus implicancias
 - Diagnosticar y solucionar problemas de rendering

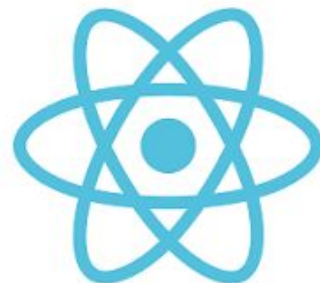
RECAP:
PRINCIPIOS BÁSICOS REACT

Como vimos en las primeras clases, React trabaja con un **flujo de reconciliación** y entiende bastante acerca de dónde y cómo ocurren los cambios en nuestra app

A su vez mantiene una versión en virtual de la misma y recordamos eso como el virtual DOM



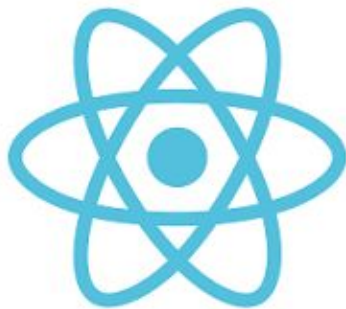
Si bien como vimos anteriormente, React **nos ayuda a escuchar eventos**, nos sigue dejando a nuestro cargo la sincronización con nuestro estado.



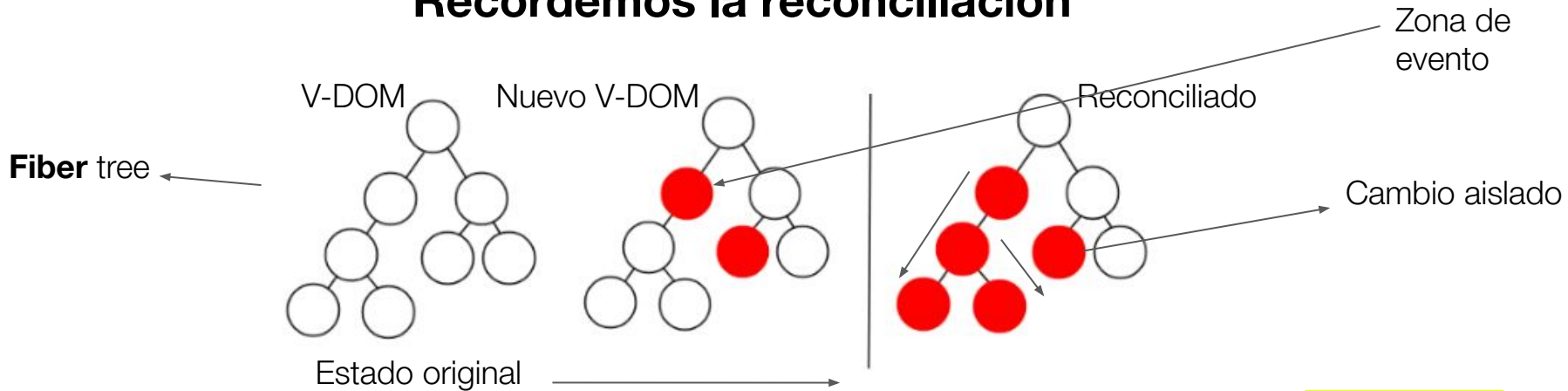
```
const ControlledInput = () => {  
  const [input, setInput] = useState('');  
  
  return <input type="text"  
    value={value}  
    onInput={(evt) => setInput(evt.target.value)} />;  
}
```

Evento sube

*El cada nuevo render causado por setInput reescribirá un input con el nuevo **value***



Recordemos la reconciliación



RENDERING CONDICIONAL

Rendering condicional

En ciclos de render puedo decidir que quiero hacer rendering de sólo algunos nodos de **un árbol** o de otro

```
function LoadingComponent() {
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    setTimeout(() => {
      setLoading(false);
    }, 10000);
  }, []);

  return <>
    { loading ? <h2>Loading</h2> : <h3>Loaded!</h3> }
  </>
}

export default function App() {
  return <LoadingComponent />
}
```

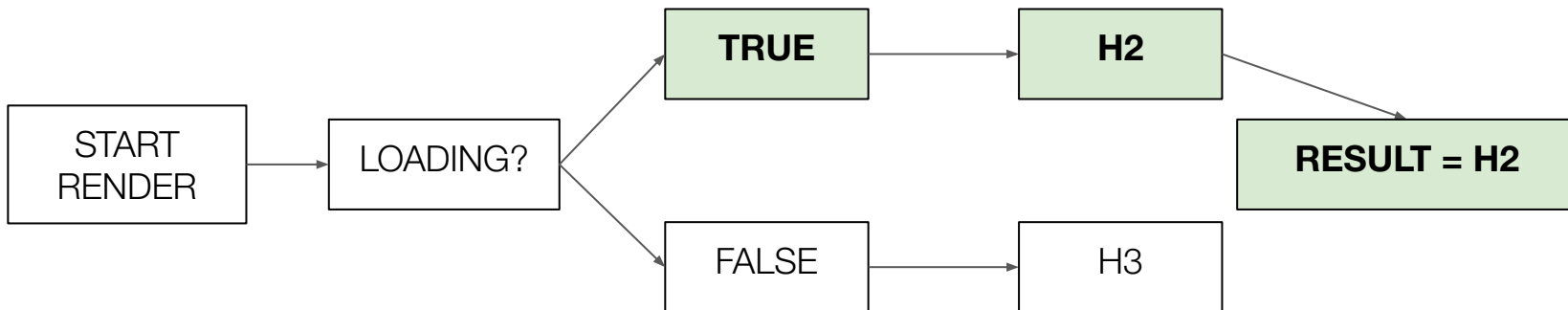
Loading

Console

React maneja el asunto con comodidad

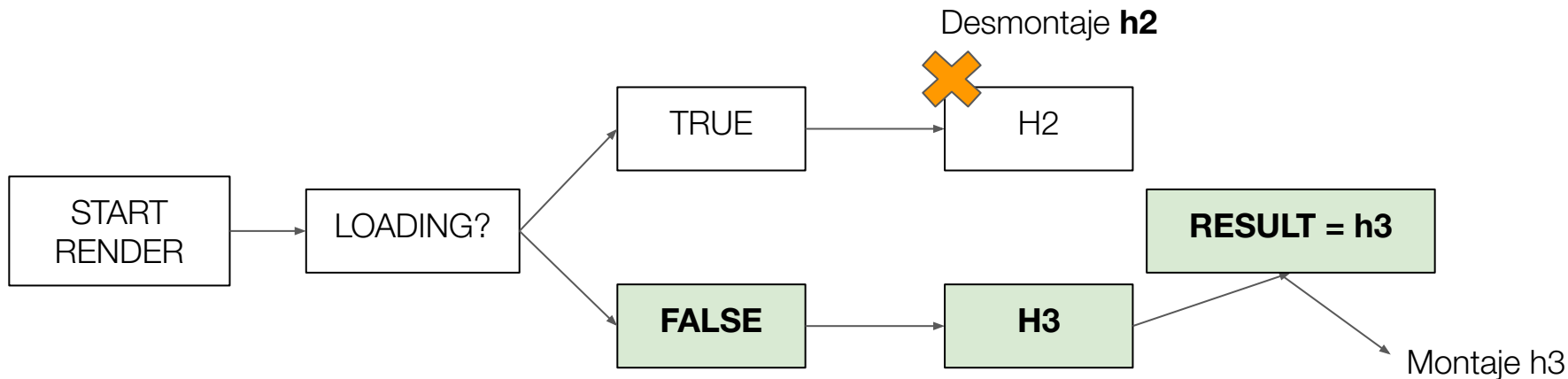
Rendering condicional

Si pensamos en **React-way** obtendremos este diagrama



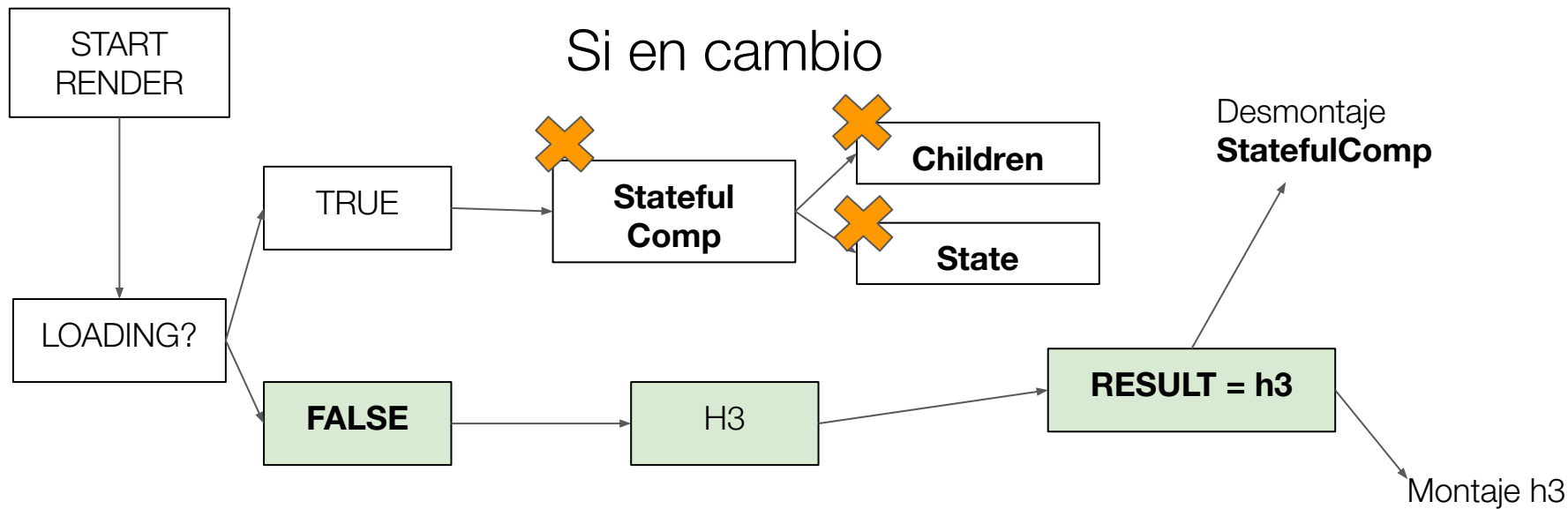
Rendering condicional

El **segundo** render se vería así



El caso es simple porque un h2 o h3 no son componentes complejos que tengan una sub-jerarquía de nodos dentro

Rendering condicional



Si por algun motivo volviese a hacer trigger de loading h3 sufriría el mismo destino de ser desmontado sin importar cuantos children tuviera



Rendering condicional

- Sirve para aparecer y desaparecer nodos del render
- Estos eventos provocan dismounting y todos los efectos que ello conlleva
- Se llamará al efecto de desmontaje y podremos detectarlo
- Podemos usar los cleanup effects para detectar algún dismounting si no sabemos con certeza si ocurre
- A veces se producen sin la intención y **causan bugs** o pérdida no intencionada del estado, dando inestabilidad

TÉCNICAS DE RENDERING CONDICIONAL

Rendering condicional #1: IF c/ return temprano

React **renderiza el resultado** del return de nuestra función y **cada return** se transforma en nuestro **nuevo árbol** de partida para próximos

```
function TextComponent({ condition }) {  
  if(condition === true) {  
    return <h2>Condition is true</h2>  
  }  
  
  return <h2>Condition is false</h2>;  
}
```

Versión JSX

vs

```
function TextComponent(_ref) {  
  var condition = _ref.condition;  
  
  if (condition === true) {  
    return React.createElement("h2", null, "Condition is true");  
  }  
  
  return React.createElement("h2", null, "Condition is false");  
}
```

¡Notar que se crean 2 **createElement** distintos!

Vanilla JS (Después de ser transpilado)

Rendering condicional #2: Inline con fragment

Tenemos un nodo base (fragment) y decidimos inline con **'condition &&'**

Versión JSX

```
function TextComponent({ condition }) {  
  return (<>  
    {condition && <h2>Condition is true</h2>}  
    {!condition && <h2>Condition is false</h2>}  
  </>);  
}
```

VS

```
function TextComponent(_ref) {  
  var condition = _ref.condition;  
  return React.createElement(React.Fragment, null, condition && React.createElement("h2",  
null, "Condition is true"), !condition && React.createElement("h2", null, "Condition is  
false"));  
}
```

Vanilla JS

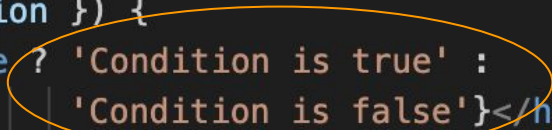
*¡Notar que se crean 2 **createElement** distintos!*

Rendering condicional #3: Inline ternary

Mantenemos el mismo nodo como padre y modificamos sus children, que en este caso son los textos, lo cual optimiza ya que no hay dismounts

Versión JSX

```
function TextComponent({ condition }) {  
  return <h2>{condition === true ? 'Condition is true' :  
    'Condition is false'}</h2>;  
}
```



VS

```
function TextComponent(_ref) {  
  var condition = _ref.condition;  
  return React.createElement("h2", null, condition === true ? 'Condition is true' :  
    'Condition is false');  
}
```

Vanilla JS

*¡Notar que se crea un único **createElement**!*

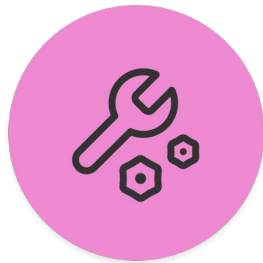
VAMOS AL CÓDIGO



CODER HOUSE

¿PREGUNTAS?





Crea un loader component

Crea en [stackblitz](https://stackblitz.com) un componente '**Loader**' dentro de la App que tenga una prop '**loading**' (boolean:true|false). Si loading es true, el componente debe mostrar "Loading..." y si es **false**, nada.

Extra: Si quieres puedes integrar algún spinner de la librería de UI que estés usando

Tiempo: 15 minutos

BREAK



CODER HOUSE

OTRAS TÉCNICAS PARA CONTROL CONDICIONAL

Conditional props: styling

El conditional rendering aplica no solo para los nodos, sino también para sus propiedades

```
function TextComponent({ condition }) {  
  return (<  
    <h2 style={{ color: !condition ? 'red' : 'green' }}>Loading...</h2>  
    </>);  
}  
  
export default function App() {  
  return <TextComponent condition={true} />  
}
```

Loading...

¡Notar que se crea un único **createElement!**

Conditional attributes: classes

Modificar clases en base a condiciones

```
function TextComponent({ condition }) {  
  return (<  
    <h2 className={condition === true ? 'greenClass' : 'redClass'}  
    >Loading...</h2>  
    </>);  
}
```

```
function TextComponent(_ref) {  
  var condition = _ref.condition;  
  return  
    React.createElement(React.Fragment, null, React.createElement("h2", {  
      className: condition ? 'greenClass' : 'redClass'  
    }, "Loading..."));  
}
```

¡Como vemos las clases se concatenan!

Conditional attributes: multi-class

```
function TextComponent({ condition, other }) {  
  return (<>  
    <h2 className={` ${condition === true ? 'redClass' : 'greenClass'} ${other || ''}`}>Loading...</h2>  
    </>);  
}  
  
export default function App() {  
  return <TextComponent condition={true} other="newClass" />  
}
```




```
<div id="root">  
  <h2 class="redClass newClass">Loading...</h2>  
</div>
```


Conditional attributes: multi-class anti-pattern

En este caso no es conveniente aplicar **condition &&**



```
function TextComponent({ condition, other }) {  
  return (<  
    <h2 className={`${condition === true ? 'redClass' : 'greenClass'} ${other && 'otherClass'}`}  
    >Loading...</h2>  
    </>);  
}  
  
export default function App() {  
  return <TextComponent condition={true} />  
}
```



```
▼ <body>  
  ▼ <div id="root">  
    <h2 class="redClass undefined">Loading...</h2> == $0  
  </div>  
  </body>
```

Conditional props/Props dinámicas

Podemos hacer
spreading de
propiedades de
manera condicional

```
function TextComponent({ condition, other }) {  
  const config = condition ? {  
    className: `redClass ${other} || ''`,  
    title: 'Title when condition is true'  
  } : {};  
  
  return (<  
    <h2 {...config}>Loading...</h2>  
    </>);  
}
```

```
export default function App() {  
  return <  
    <TextComponent condition={true} />  
  >
```

Loading...

Console

Preview (local)

Console was closed

```
<body>  
  <div id="root">  
    <h2 class="redClass " title="Title when condition is true">  
      Loading...</h2> == $0  
    </div>  
  </body>
```

VAMOS AL CÓDIGO

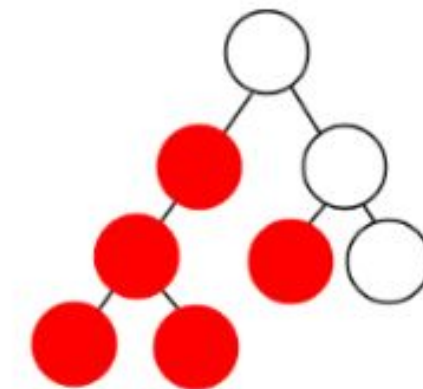


CODER HOUSE

RENDER OPTIMIZATION

En cada cambio*, **React** hace rendering top-down desde el lugar donde se produjo el cambio de estado de manera recursiva, hacia las hojas.

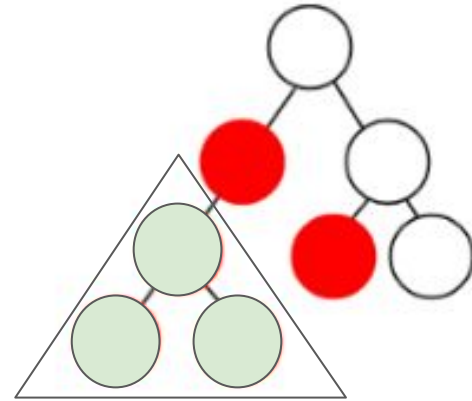
Muchas veces sabemos que en realidad nada debería cambiar en algunos componentes montados.



* Recordemos que los sources de cambio son props & state

Esto quiere decir que **podemos salvarle** a React ese trabajo, si:

1. El componente es **puro**
2. Tenemos la certeza de que **las mismas props producen siempre el mismo render**
3. Sabemos que es muy caro de realizar, una lista larga, compleja, etc.



Memoizing

MEMO (IZATION)

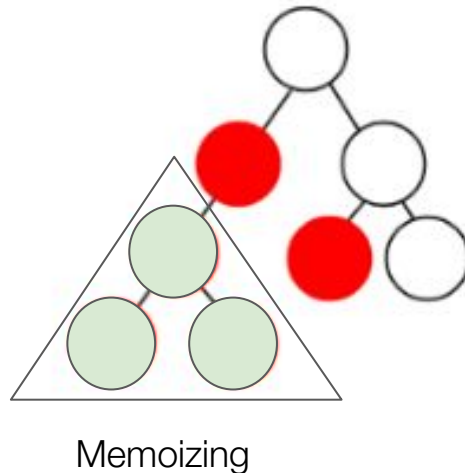
Para lograrlo, solo necesito envolver mi componente en un memo:

```
function Comp() { return <> }
```

```
React.memo(Comp, fn*)
```

Donde `fn` es la función comparadora

```
*fn = (prevProps, nextProps) => true|false  
true => usar memo! / false => re-render!
```



React Memo: Configuración default

Por default se usa comparación superficial (shallow)

```
const ListItem = React.memo(({ item }) => {  
  |   return <li>{item.id}</li>  
  | }  
  | })
```

Previo: { id: 1, name: 'item 1 name' }

Nuevo: { id: 1, name: 'item 1 name' }

Solo se invocará el render al montar

React Memo: Validación manual

Puedo decirle a React que solo haga re-rendering cuando una propiedad específica cambie

```
const ListItem = React.memo(({ item }) => {  
  console.log('Rendering item');  
  return <li>{item.id}</li>  
}, (oldProps, newProps) => oldProps.item.modifyDate === newProps.item.modifyDate)
```

*En este caso solo cambiará cuando **modifyDate** sea distinta*

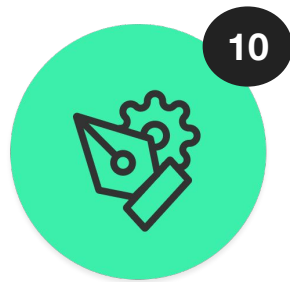


React memo

- Sirve para ahorrar renders costosos de los cuales podemos preveer el resultado en base al análisis de sus props
- No es necesario **ni recomendado** usarlo en todos lados o en componentes simples
- Útil en listas largas y determinadas que se tienen un re-rendering frecuente pero que no modifica sus props
- Si te interesa expandir, ¡también hay un hook para [memoizar](#) otros cálculos aparte de renders!

¿PREGUNTAS?





Cart view

Usa el nuevo **CartContext** para hidratar tu nueva vista `‘/cart’` y agrupa los ítems para que el user vea sus ítems seleccionados. De no haber ítems muestra un mensaje, de manera condicional, diciendo que no hay ítems y un react router **Link** para que pueda volver a la home para buscar y comprar algo.

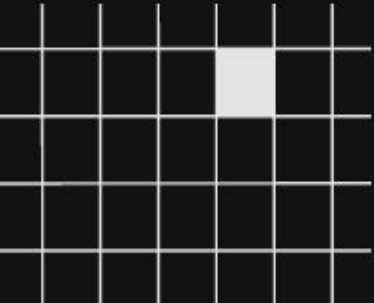
Formato de entrega: GIF mostrando que puedes agregar items y verlos en /cart

Tiempo: 30 minutos



¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Rendering condicional
 - Memoization
 - Estrategias de optimización
- 



OPINA Y VALORA ESTA CLASE