

Tema 3 - Interfaces complejas mediante composición

Contenido

- Composición
- Formularios
- Acceso al DOM y ciclo de vida de un componente



Composición



Composición

- Separación en componentes
- Diseño del estado
- Flujo de datos entre componentes
- Roles

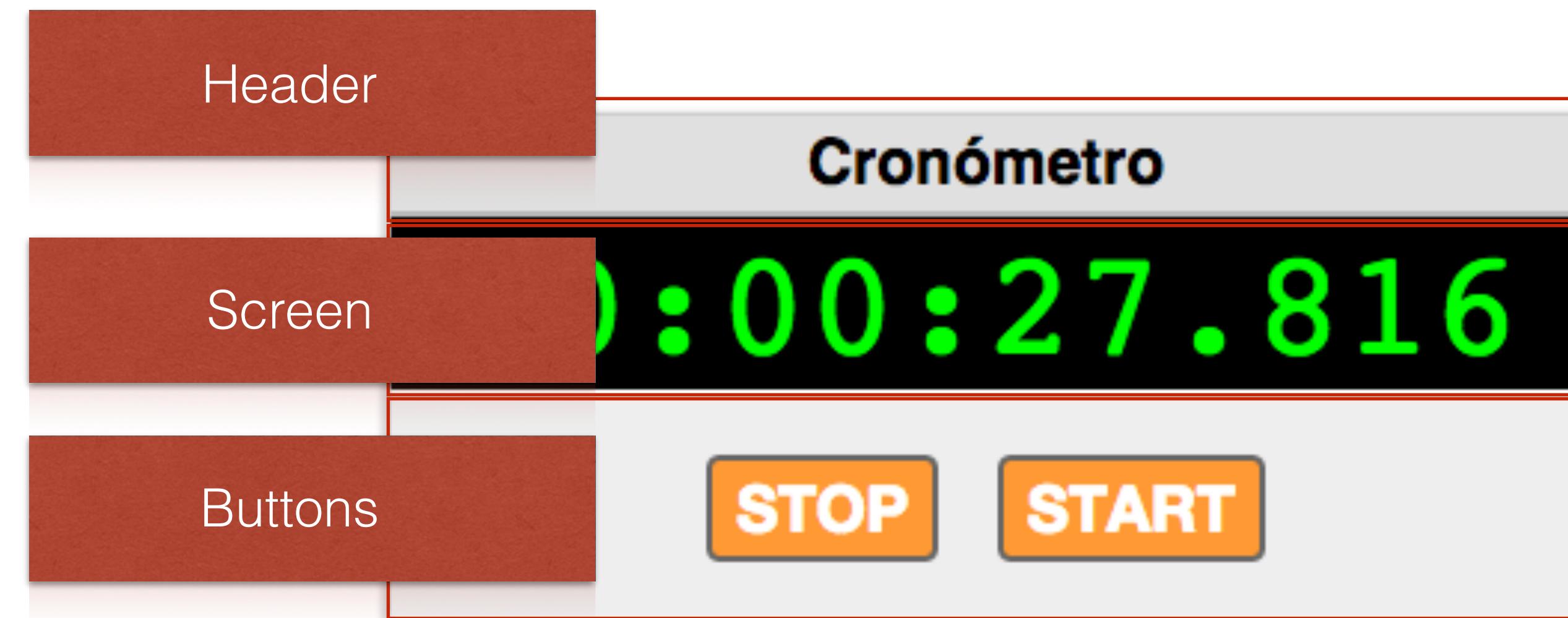
Separación en componentes

- ¿Cómo identificamos los componentes?
- **Separación de intereses**
- Responsabilidad única

Ejemplo



Ejemplo



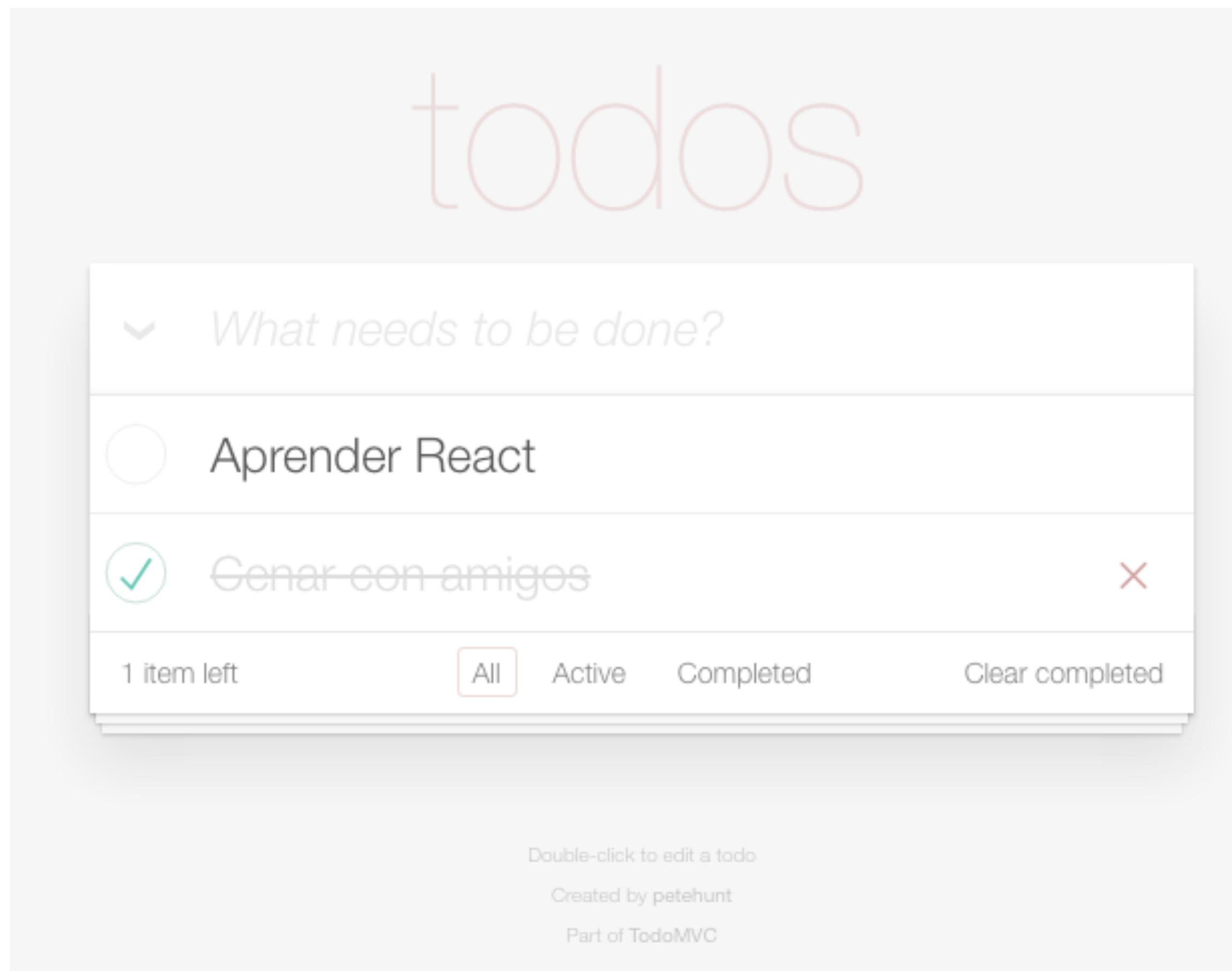
Diseño del estado

- Interactividad implica estado
- Mínimo estado posible
- No guardar información calculable

Diseño del estado

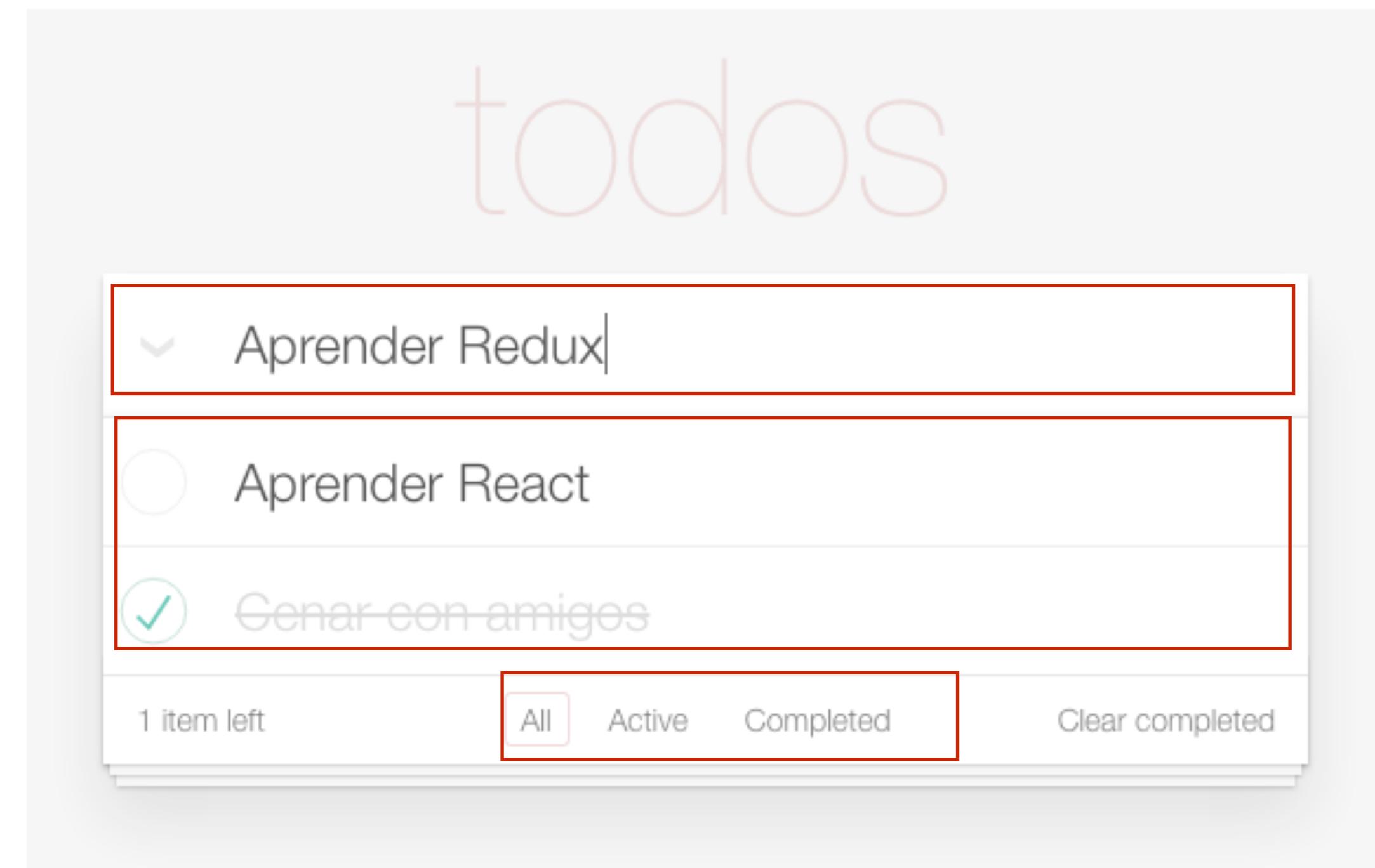
- Si no cambia en el tiempo, **no** es estado
- Si lo puedo calcular en base a props, no es estado

Ejemplo



Ejemplo

- Texto nuevo todo
- Lista de tareas
- Filtro actual



Ejemplo

```
{  
  todos: [  
    {  
      text: 'Aprender React',  
      completed: false  
    },  
    {  
      text: 'Cenar con amigos',  
      completed: true  
    }  
  ],  
  filter: 'all',  
  newTodo: 'Aprender Redux'  
}
```

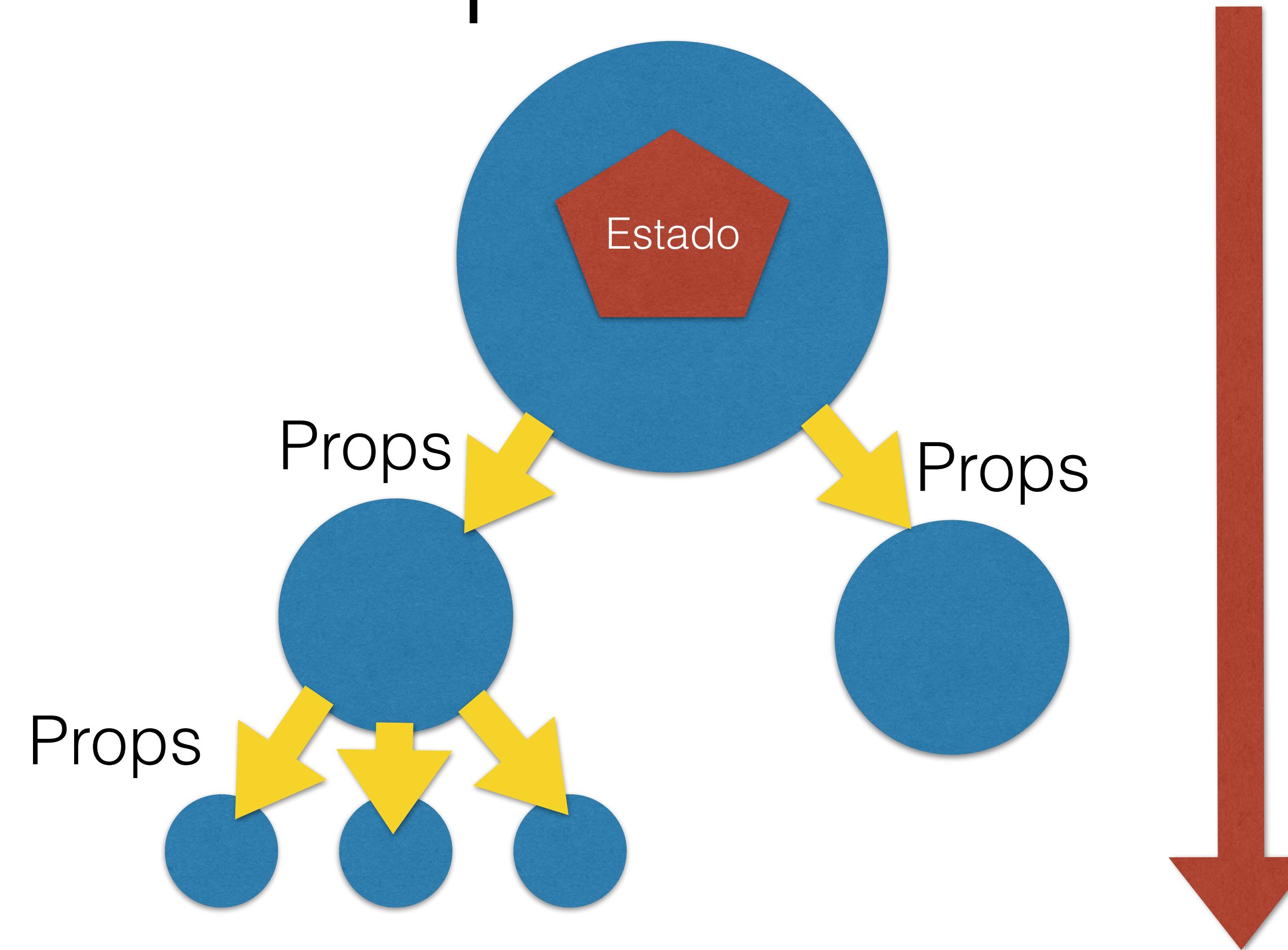
Diseño del estado

- Donde haya estado interno, habrá mutaciones
- Mejor centralizar y abstraer hacia arriba

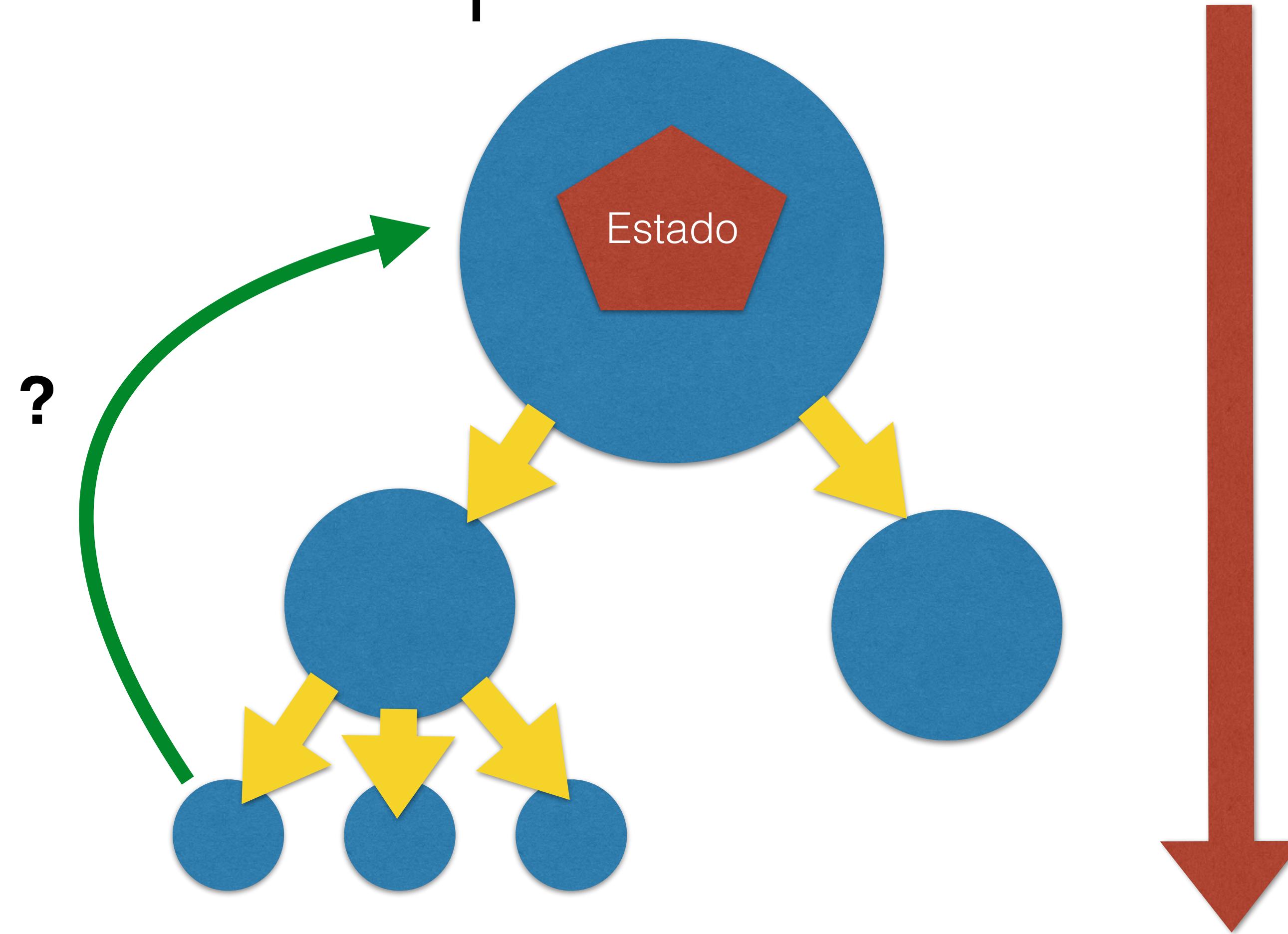
Flujo de datos entre componentes

- Interfaz dividida y estado de la aplicación
- **props** de un componente es su interfaz con mundo exterior

Flujo de datos entre componentes



Flujo de datos entre componentes



Flujo de datos inverso

- JSX es Javascript
- Podemos aceptar funciones como props
- El padre configura al hijo con "callbacks"
- Los hijos llamarán a esas funciones en respuesta a eventos

Ejemplo

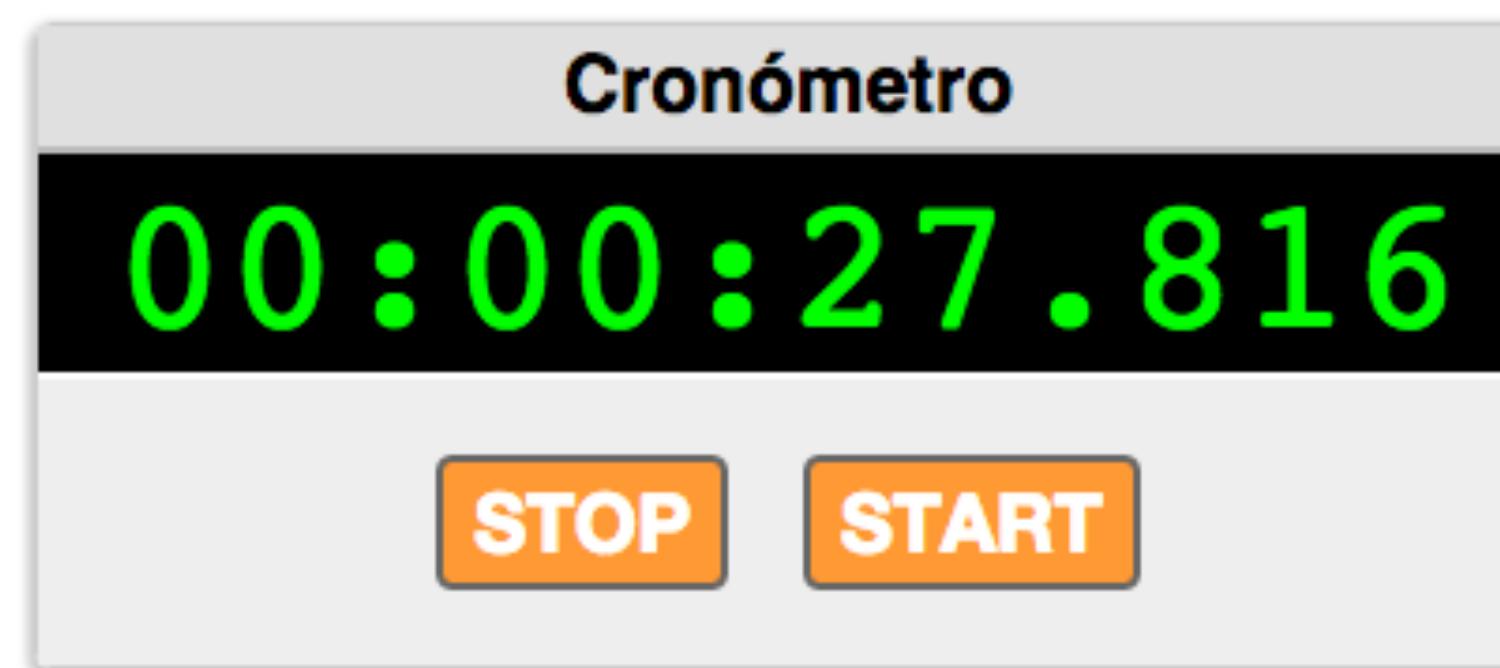
```
class Buttons extends Component {  
    render() {  
        return (  
            <div className="actions">  
                <button onClick={ this.props.onCancel }>Cancelar</button>  
                <button onClick={ this.props.onAccept }>Aceptar</button>  
            </div>  
        ) ;  
    }  
}  
  
Buttons.propTypes = {  
    → onCancel: React.PropTypes.func,  
    → onAccept: React.PropTypes.func  
}
```

Ejemplo

```
class RegisterForm extends Component {
  constructor() {
    super();
    this.handleAccept = this.handleAccept.bind(this);
    this.handleCancel = this.handleCancel.bind(this);
  }
  ➔ handleAccept(e) {
    console.log("Aceptar en Buttons!");
  }
  ➔ handleCancel(e) {
    console.log("Cancelar en Buttons!");
  }
  render() {
    return (
      <div className="register-form">
        <form>
          { /* ...otros componentes... */ }
        </form>
        <Buttons
          onAccept={this.handleAccept}
          onCancel={this.handleCancel}
        />
      </div>
    );
  }
}
```

Ejercicio: cronómetro

- Vamos a practicar con props, estado interno y eventos
- Requisitos
 - Botón START inicia el temporizador
 - Botón STOP lo detiene en el primer clic y lo reinicia a 0 en el segundo.



Ejercicio: cronómetro

- **Plantilla HTML**
`/ejercicios/tema3/src/templates/cronometro.html`
- **Estilos CSS**
`/ejercicios/tema3/dist/index.css`
Añadir la hoja de estilos a **index.html**.
- Funciones auxiliares para manipular el tiempo
`/ejercicios/tema3/src/lib/utils.js`

Cronómetro: pasos a seguir

- Crear un componente raíz (aplicación) Cronometro
- Identificar componentes
- Diseñar estado e interactividad en Cronometro
- Definir flujos de datos (props y callbacks)

Roles de componentes

- React: todo son componentes
- ¿Podemos agruparlos por función o por rol?

Roles de componentes

- Por su **rol**, distinguimos dos grupos de componentes
- **Componentes presentacionales** / Dumb components
- **Contenedores** / Controladores / Smart components

Composición: presentacionales

- Son abstracciones sobre HTML
- Dependen exclusivamente de **props**
- Muy poca o ninguna lógica propia
- Muy reutilizables, muy configurables



Composición: contenedores

- **render** de N presentacionales
- Estado
- Lógica
- Orquestación



Composición avanzada

- JSX está basado en XML (como HTML)
- Un nodo puede tener otros nodos como hijos
- <p>Hola <**strong**>mundo</**strong**></p>

Composición avanzada

```
<ModalWindow title='Cancelar pedido'>
  <p>¿Seguro que quiere cancelar el pedido?</p>
  <Buttons
    onCancel={ this.handleCancel }
    onAccept={ this.handleAccept } />
</ModalWindow>
```

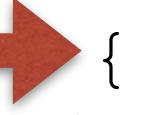
Composición avanzada



There is no HTML

Composición avanzada

```
<ModalWindow title='Cancelar pedido'>
  <p>¿Seguro que quiere cancelar el pedido?</p>
  <Buttons
    onCancel={ this.handleCancel }
    onAccept={ this.handleAccept } />
</ModalWindow>
```

```
//ModalWindow
render() {
  return (
    <div className='modal'>
      <h1 className='modal-title'>{ this.props.title }</h1>
       { this.props.children }
    </div>
  )
}
```

Resumen: la manera React

Los componentes son cajas negras que "cableamos" mediante sus props.



Formularios

Contact Form

First name	Last name
Email address	Telephone number
Website url	Select department
Enter message or comment	
Hint: Don't be negative or off topic	
Enter captcha	46FBUJ C
Cancel	Submit Form

Formularios

- Los controles de formulario HTML son **especiales**
- Son mutables mediante interacciones de usuario
- **render()** - representación en un momento dado

Formularios

```
class TextInput extends Component {  
  render() {  
    return (  
      <input type="text" value="Introduce tu nombre" />  
    );  
  }  
}
```

Si intentamos escribir en esa caja de texto, no pasará nada
¿Por qué?

Formularios - onChange



There is no HTML

Formularios

```
class TextInput extends Component {  
  render() {  
    return (  
      <input type="text" value="Introduce tu nombre" />  
    );  
  }  
}
```

Porque la **prop value** dice que, invariablemente, el valor de ese INPUT sea “Introduce tu nombre”

Formularios - props especiales

- value, checked, selected
- onChange

Formularios - value

- **value** - Establece el valor en:

```
<input type="text" .../>
```



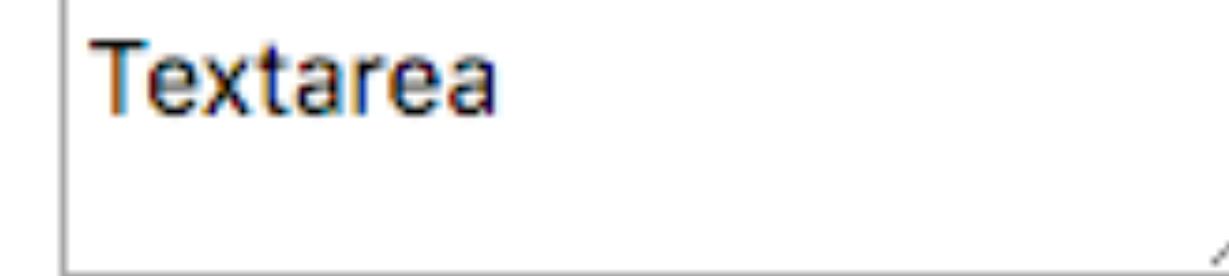
```
<input type="password" .. />
```



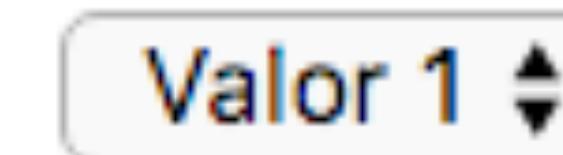
```
<input type="radio" .. />
```



```
<input type="checkbox" .. />
```



```
<textarea .. />
```



```
<select />
```

Formularios - checked

- **checked** - (Boolean) recupera/establece si están activos:

```
<input type="checkbox" .../>
```

```
<input type="radio" .. />
```

Formularios - selected

- **selected** - (Boolean) recupera/establece si están seleccionados los elementos **option** de un desplegable:

```
<select>
  <option value="1">Uno</option>
    <option value="2">Dos</option>
  </select>
```

Formularios - onChange

- Para saber cuándo el usuario modifica el valor, evento **onChange**
- Funciona en **todos** los controles de formulario

Formularios

- Dos formas de trabajar con formularios en React:
con componentes **controlados** o **no controlados**
- Los controlados dependen sólo de props
- Los no controlados tienen su estado interno y
funcionan como en HTML

Formularios

- Controlado - establecemos **value** (checked | selected)
- No controlado - sin **value** (checked, selected)

Formularios - no controlados

- Dejamos que el propio control gestione su estado
- Nos interesa el valor cuando se **envía** el formulario
- ¿Cómo accedemos a los campos?

Usuario:

foo

Password:

...

Iniciar sesión

Refs

- React gestiona el DOM por nosotros
- Podemos generar una referencia que se actualice entre **render()** sucesivos

```
<button ref="miboton">Click me</button>
```

- **this.refs.miboton** = nodo del DOM
- Si React elimina ese nodo = *undefined*

Ref callback

- La prop **ref** acepta una función
- A la que llamará con el DOMNode como parámetro

```
<input type='text' ref={ node => this._user = node } />
```

- Muy cómodo con Arrow Function
- Guardamos el nodo donde queramos o actuamos sobre él

Ejemplo no controlado

```
handleSubmit(e) {
  e.preventDefault();
  console.log('Registrando', this._user.value, this._pwd.value)
}
render() {
  return (
    <div>
      <form onSubmit={ this.handleSubmit }>
        <p>
          Usuario: <br />
          <input type='text' ref={ x => this._user = x } />
        </p>
        <p>
          Password:<br />
          <input type='password' ref={ x => this._pwd = x } />
        </p>
        <p><button type='submit'>Iniciar sesión</button></p>
      </form>
    </div>
  )
}
```

Ejemplo no controlado

```
handleSubmit(e) {
  e.preventDefault();
  console.log('Registrando', this._user.value, this._pwd.value)
}
render() {
  return (
    <div>
      <form onSubmit={ this.handleSubmit }>
        <p>
          Usuario: <br />
          <input type='text' ref={ x => this._user = x } />
        </p>
        <p>
          Password:<br />
          <input type='password' ref={ x => this._pwd = x } />
        </p>
        <p><button type='submit'>Iniciar sesión</button></p>
      </form>
    </div>
  )
}
```

Formularios: componentes controlados

- Al establecer **value** indicamos que queremos el control total
- Somos responsables de guardar el estado y actualizar sus props
- Recuerda: **onChange** funciona en todos los controles

Formularios: componentes controlados

```
class LoginFormControlled extends Component {
  constructor() {
    super()
    this.state = {
      usuario: '',
      password: ''
    }
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  handleSubmit(e) {
    e.preventDefault();
    const { usuario, password } = this.state;
    console.log('Iniciando sesión para', usuario, password);
  }
  handleChange(e) {
    this.setState({
      usuario: e.target.value
    });
  }
  render() {
    const { usuario, password } = this.state;
    return (
      <div>
        <form onSubmit={ this.handleSubmit }>
          <p>Usuario: <br />
            <input type='text' id='usuario' value={ usuario } onChange={ this.handleChange } />
          </p>
          <p>Password:<br />
            <input type='password' id='password' value={ password } />
          </p>
          <p><button type='submit'>Iniciar sesión</button></p>
        </form>
      </div>
    )
  }
}
```

Formularios: componentes controlados

```
class LoginFormControlled extends Component {
  constructor() {
    super()
    this.state = {
      usuario: '',
      password: ''
    }
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }
  handleSubmit(e) {
    e.preventDefault();
    const { usuario, password } = this.state;
    console.log('Iniciando sesión para', usuario, password);
  }
  handleChange(e) {
    this.setState({
      usuario: e.target.value
    });
  }
  render() {
    const { usuario, password } = this.state;
    return (
      <div>
        <form onSubmit={ this.handleSubmit }>
          <p>Usuario: <br />
            <input type='text' id='usuario' value={ usuario } onChange={ this.handleChange } />
          </p>
          <p>Password:<br />
            <input type='password' id='password' value={ password } />
          </p>
          <p><button type='submit'>Iniciar sesión</button></p>
        </form>
      </div>
    )
  }
}
```

Formularios: componentes controlados

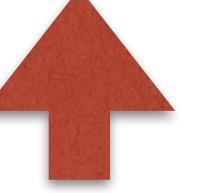
- ¿Y hay que hacer un método **handleXXX** por cada control del formulario?
- Sí y no
- Podemos hacer magia con Babel :)

Formularios: componentes controlados

```
handleChange(e) {
  this.setState({
    [e.target.id]: e.target.value
  ) ;
}
```

Formularios: componentes controlados

```
handleChange(e) {  
  this.setState({  
    [e.target.id]: e.target.value  
  }) ;  
}
```



```
<input id="usuario" onChange={ this.handleChange } />
```

Thug Life



Nombres de propiedad dinámicos

Ejercicio: Todos

- Hagamos la típica aplicación de "to-dos"
- Input para añadir una nueva tarea, click en una tarea existente para completada / no completada

Todos

Hey!

- let's go!
- ~~he~~
- ~~hey~~

Ver: Todos | [Activos](#) | [Completados](#)

Ejercicio: Todos

- Maqueta HTML

/ejercicios/tema3/src/templates/todos.html

Ejercicio: Buscador

- Buscador de personajes de Juego de Tronos

Buscador Juego de Tronos

No Spoilers

Actor / personaje	Familia	Aparece en temporada	
<input type="text"/>	Todas	<input type="checkbox"/> 1 <input type="checkbox"/> 2 <input type="checkbox"/> 3 <input type="checkbox"/> 4 <input type="checkbox"/> 5 <input type="checkbox"/>	
<input type="checkbox"/> Sólo personajes vivos			
Personaje	Actor	Nº Ep	Vivo
Eddard Stark	Calvin Hobbs	45	Sí
Eddard Stark	Calvin Hobbs	45	Sí
Eddard Stark	Calvin Hobbs	45	Sí

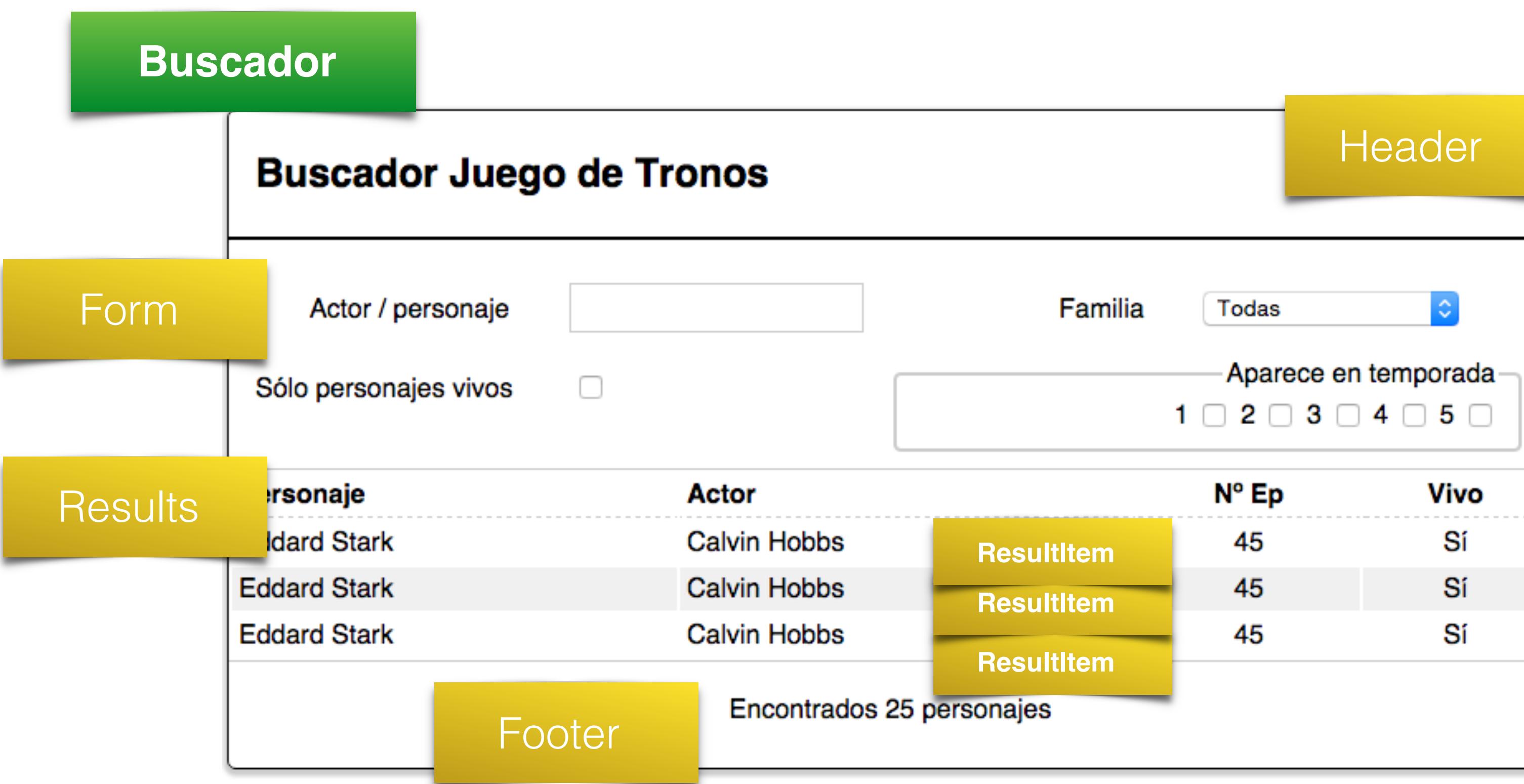
Encontrados 25 personajes

Ejercicio: Buscador

- Actualiza los resultados en vivo, según se modifican los parámetros de búsqueda (**onChange**)
- Los datos en JSON:
/ejercicios/tema3/src/data/got.js
- Maqueta HTML
/ejercicios/tema3/src/templates/buscador.html
- Esqueleto en el repositorio:
/ejercicios/tema3/src/components/buscador/

Ejercicio: Buscador

- Componente padre: **Buscador** (estado interno y lógica)
- Hijos: **Header, Form, Results...**



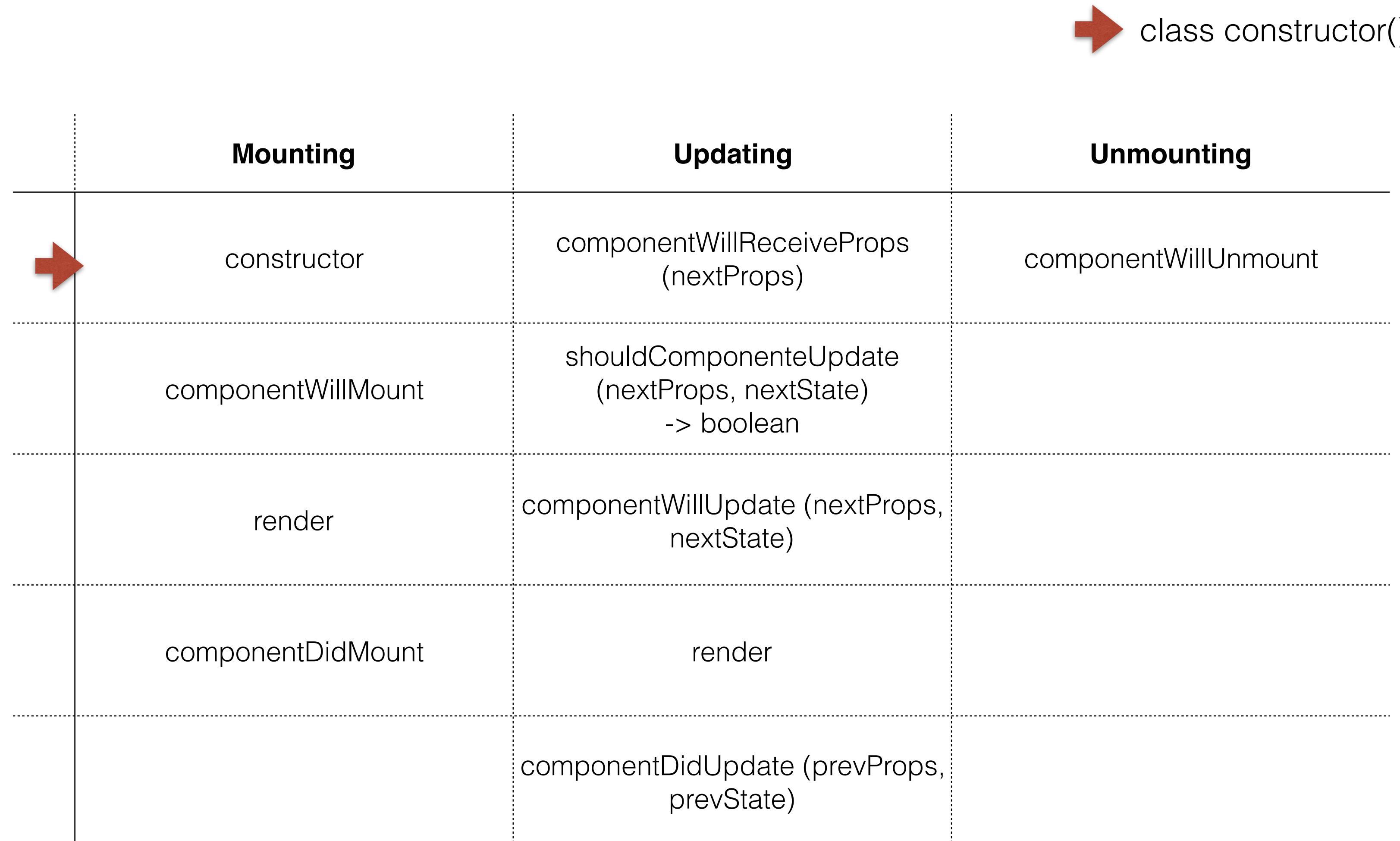
Acceso al DOM

- Nuestro código Javascript genera al final nodos de Virtual DOM
- React nos abstracta del DOM real
- ¿Cuándo está el DOM listo?
- ¿Cuándo se va a eliminar un componente?
- ¿Cuándo el padre ha pasado nuevas **props**?

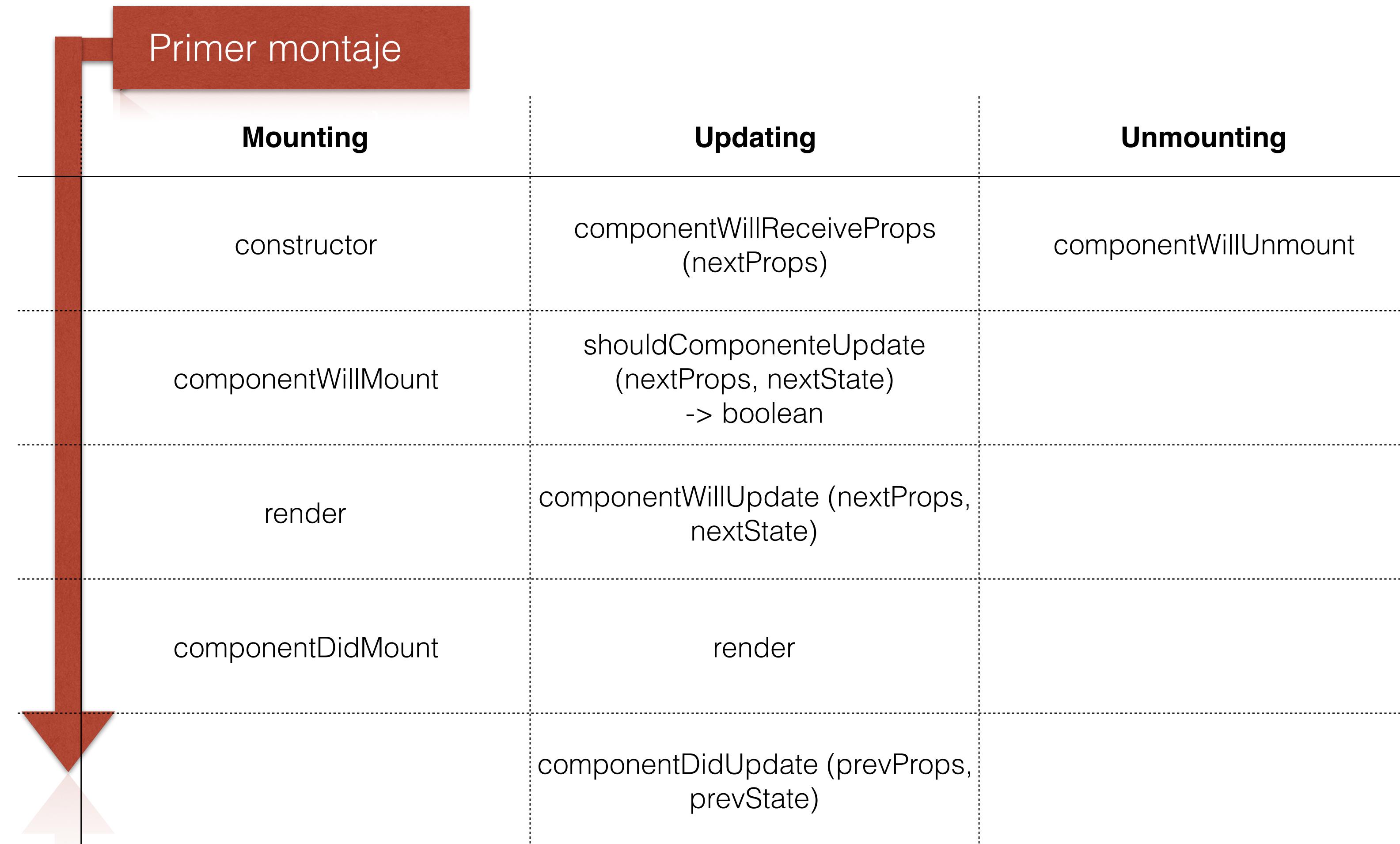
Ciclo de vida de un componente

- Existen 3 momentos en el ciclo de vida de un componente:
 1. **mounting** (creación, primer montado en DOM)
 2. **updating** (actualización props o estado)
 3. **unmounting** (destrucción, desmontar de DOM)

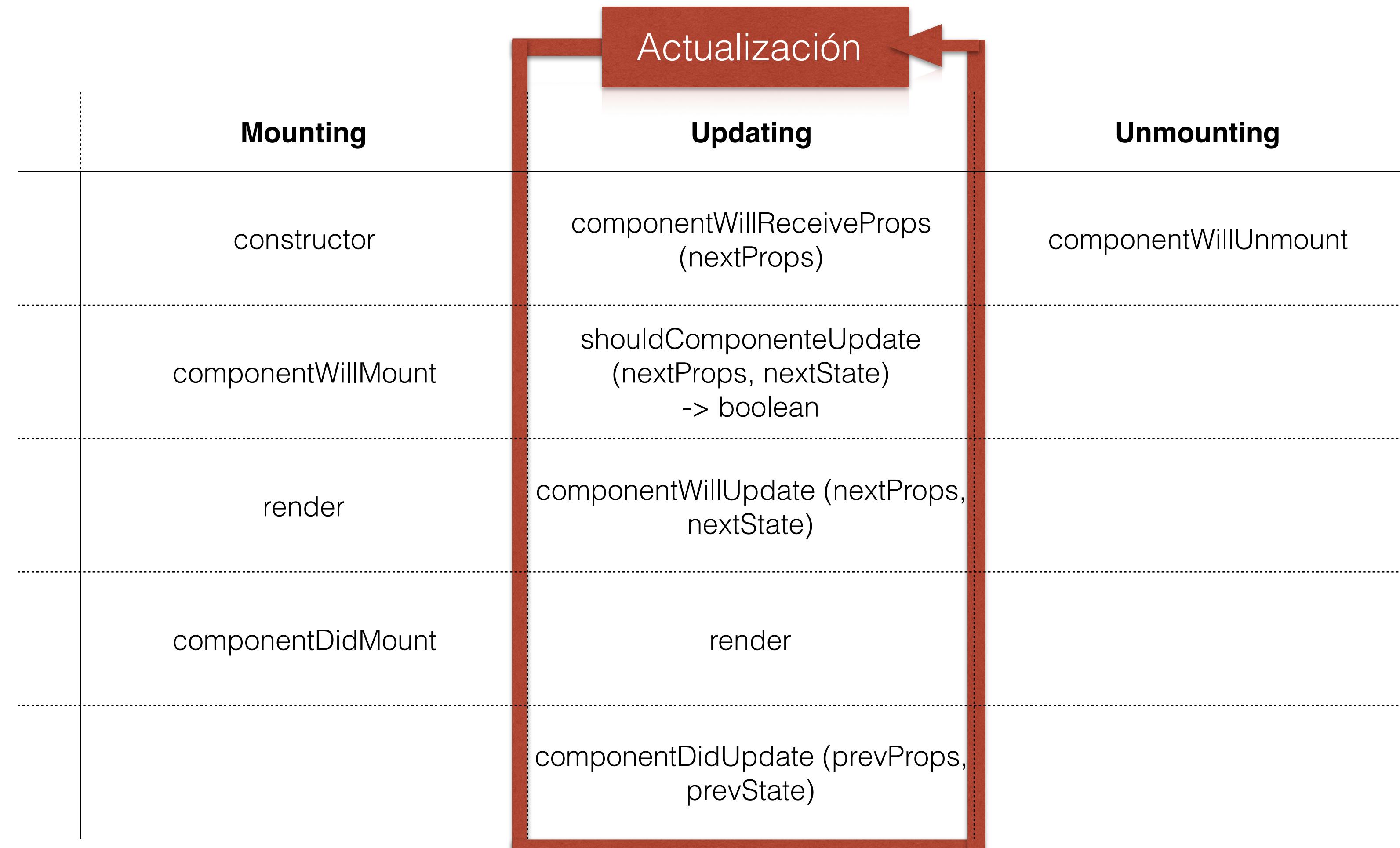
Ciclo de vida de un componente



Ciclo de vida de un componente



Ciclo de vida de un componente



Ciclo de vida de un componente

	Mounting	Updating	Unmounting	Destrucción
	constructor	componentWillReceiveProps (nextProps)	componentWillUnmount	
	componentWillMount	shouldComponentUpdate (nextProps, nextState) -> boolean		
	render	componentWillUpdate (nextProps, nextState)		
	componentDidMount	render		
		componentDidUpdate (prevProps, prevState)		

Ciclo de vida de un componente

	Mounting	Updating	Unmounting
	constructor	componentWillReceiveProps (nextProps)	componentWillUnmount
	componentWillMount	shouldComponentUpdate (nextProps, nextState) -> boolean	
	render	componentWillUpdate (nextProps, nextState)	
	componentDidMount	render	
		componentDidUpdate (prevProps, prevState)	

Ciclo de vida de un componente

- El flujo en React siempre es unidireccional en la etapa de actualización
- **render()** es una función pura
- Dadas las mismas props y mismo estado devuelve exactamente lo mismo

Optimización

- Virtual DOM **ejecuta** render para sacar las diferencias
- **shouldComponentUpdate** es el método con el que podemos *cancelar* esa llamada a render
- Recibe las próximas *props* y próximo *state*
- Ahorramos la ejecución de render (y de sus componentes hijos)

Optimización

- React facilita una utilidad para esto: **shallowCompare**
- **npm install -S react-addons-shallow-compare**

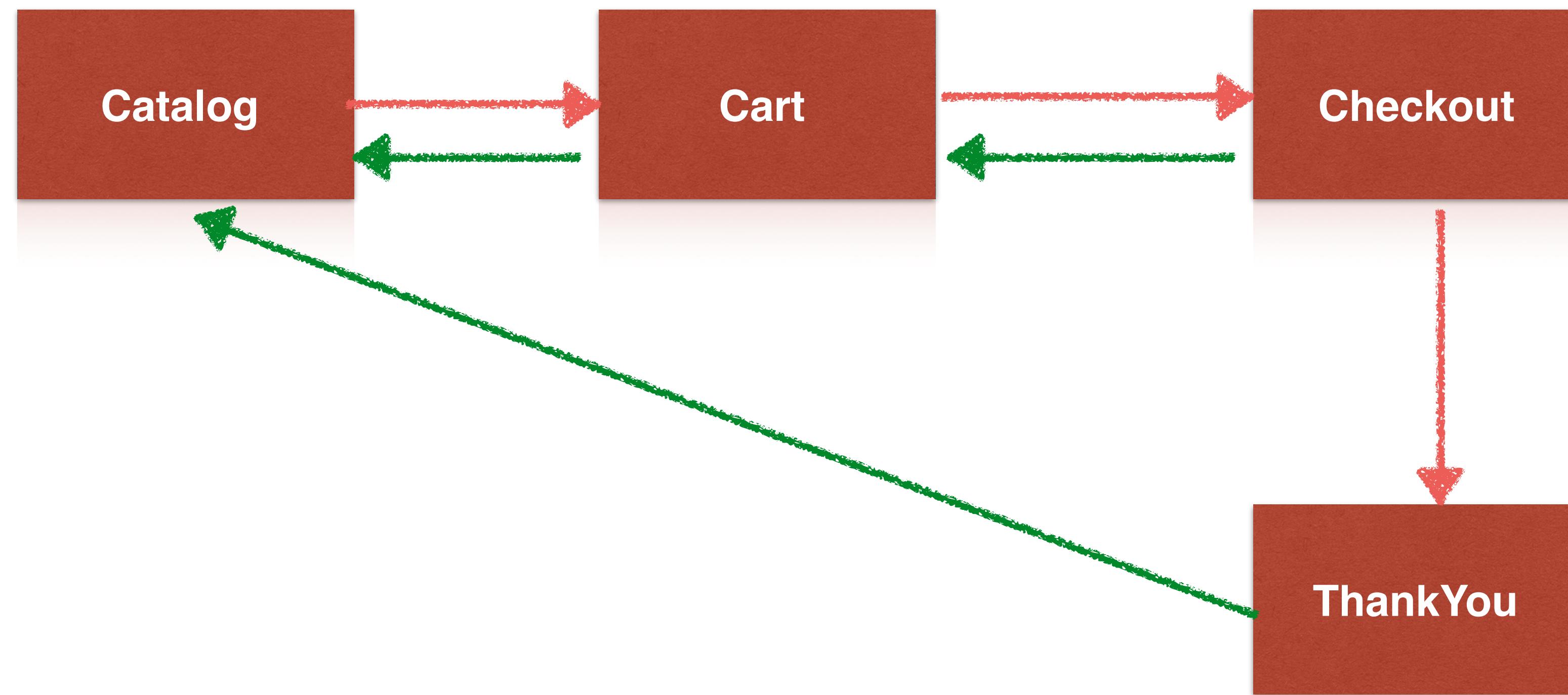
```
→ import React, { Component } from 'react';
→ import shallowCompare from 'react-addons-shallow-compare';

export default class MyComponent extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    //comparación "superficial"
    → return shallowCompare(this, nextProps, nextState)
  }
  render() {
    const { nombre, apellido, edad } = this.props;
    return (
      <div>
        Nombre: { nombre }<br />
        Apellido: { apellido }<br />
        Edad: { edad }
      </div>
    )
  }
}
```

Ejercicio - Ecommerce

- Mini tienda que contiene diferentes pantallas:
 - Catálogo - se muestran la **lista** de productos y se pueden añadir al carrito
 - Carrito - se muestran la **lista** de los productos escogidos, se manipula su cantidad y se vuelve al catálogo o se va al checkout
 - Checkout - se piden datos del usuario, se **validan** y, si es correcto, se va a la página de gracias
 - Confirmación - se muestra un mensaje de confirmación y se puede volver al Catálogo.

Ejercicio - Ecommerce



Ejercicio - Ecommerce

- Pantalla "visible" sin router
- Una propiedad **page** en el estado del contenedor raíz
- **setState({ page: 'xxxx' })** para navegar
- Su valor determina qué contenedor mostrar

Ejercicio - Ecommerce

```
render() {
  return (
    <div className="shopping-cart">
      { this.getPageComponent(this.state.page) }
    </div>
  );
}
```

Ejercicio - Ecommerce

```
getPageComponent (page) {  
    switch (page) {  
        case 'catalog':  
            return <Catalog ... />;  
        case 'cart':  
            return <Cart ... />  
        case 'checkout':  
            return <Checkout ... />;  
        case 'thank-you':  
            return <ThankYou ... />;  
    }  
}
```

Ejercicio - Ecommerce

- Plantilla HTML disponible
/ejercicios/tema3/src/plantillas/shoppingcart.html
- Datos del catálogo
/ejercicios/tema3/src/data/catalog.js
- Esqueleto de componentes
/ejercicios/tema3/src/componentes/ecommerce