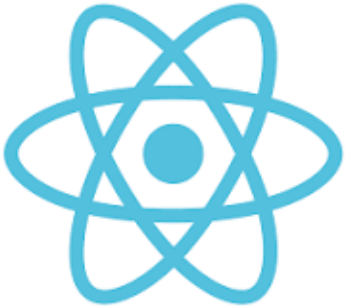
React

COMPONENTES II: INTRODUCCIÓN

Props

State



DOM Sync

Lifecycle

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Propiedades/Props

Props:

- No están limitadas a ser valores fijos como: 1 / “Alexis” / true
- Pueden ser lo que sea:
  - **Valores comunes**
    - num, bool, array, objetos
  - **Funciones**
  - **Componentes** Si los componentes son funciones, ¡entonces puedo pasar componentes! ;)
  - **Valores inyectados por librerías**
    - location, rutas, traducciones

Tarea de React - Componente Contador

- Crear un nuevo componente dentro de la carpeta SRC llamado Contador
- El Contador debe de ser un Functional Component
- El contenido del Contador debe de ser:

```
<h1>Contador</h1>
<h2> { valor } </h2>
```

- Donde "valor" es una propiedad enviada desde el padre hacia el componente Contador (Debe ser númerica validada con PropTypes)
- Renderizar el componente Contador (no se olviden del value que debe de ser un número)
- No tener errores ni warnings (Cualquier warning no usado, comentar el código)

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Estado/State: Class based

El estado en las clases era "*más simple*" de mantener, porque las clases en sí tienen un contexto propio (this.state) persistente

Class based components

componentes basados en clases

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      name: 'React'
    };
  }

  render() {
    return (
      <div>
        <Hello name={this.state.name} />
      </div>
    );
  }
}
```

Hello React!

Utilizando **this.setState** se podía guardar en **this.state** , que persiste entre renders, porque la clase se crea al montar y se destruye al desmontar

Class based components

componentes basados en clases

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      name: 'ReactClass'
    };
  }

  updateName = () => {
    this.setState({ name: 'ReactFunction'});
  }

  render() {
    return (
      <div onClick={this.updateName}>
        <Hello name={this.state.name} />
      </div>
    );
  }
}
```

Hello ReactClass!

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Estado/State: Function based

El problema es que **las funciones viven únicamente durante el tiempo que son ejecutadas** .

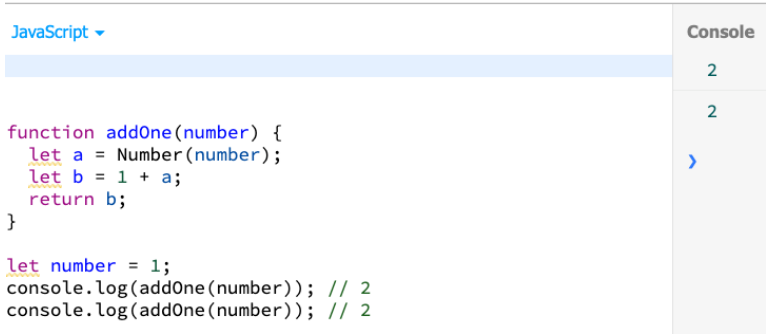
Function based components
componentes basados en funciones

```
function App() {
  const state = 'Esto morirá al finalizar la función :(';

  return <p>{state}</p>
}
render(<App />, document.getElementById('root'));
```

Esto deriva de la manera en la que ocurren las cosas en JS

Al terminar la ejecución de **addOne (num), a y b** serán puestas a disposición del **garbage collector**

Function based components
componentes basados en funciones



Todas las constantes o variables que declare para “intentar” mantener el estado, morirán y serán reiniciadas en cada render

Function based components
componentes basados en funciones

```
function App() {
  const state = 'Esto morirá al finalizar la función :(';

  return <p>{state}</p>
}
render(<App />, document.getElementById('root'));
```

Cada evento que ocurra cumpliendo ciertas características invocará el completo de la función una vez por cada re-render

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

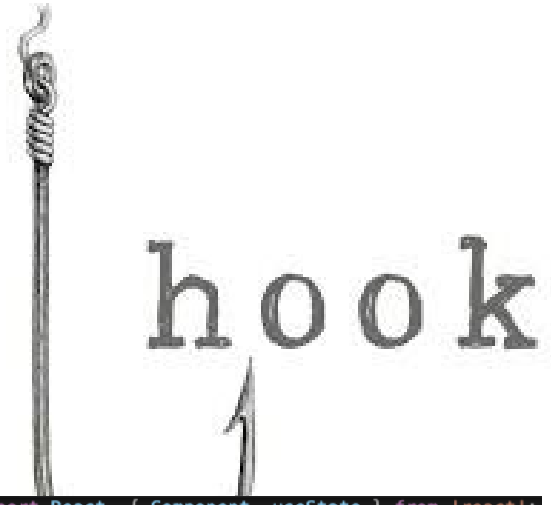
State Hook

State hook

Antes

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

class ClassApp extends Component {
  constructor() {
    super();
    this.state = {
      name: 'ReactClass'
    };
  }

  updateName = () => {
    this.setState({ name: 'ReactFunction' });
  }

  render() {
    return (
      <div onClick={this.updateName}>
        <Hello name={this.state.name} />
      </div>
    );
  }
}

render(<ClassApp />, document.getElementById('root'));
```

Hello ReactClass!




```
import React, { Component, useState } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function FnApp() {
  const [name, setName] = useState('ReactClass');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

Simplificado con hooks

State hook: Estructura básica

Se usan de la siguiente manera:

```
useState([valorInicial])
```

Devuelven un array:

```
[0] => valor (ref)
[1] => setName (fn)
```

```
import React, { Component, useState } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}

render(<App />, document.getElementById('root'));
```

State hook: Estructura básica

Los declaramos con **spread syntax** para simplificar

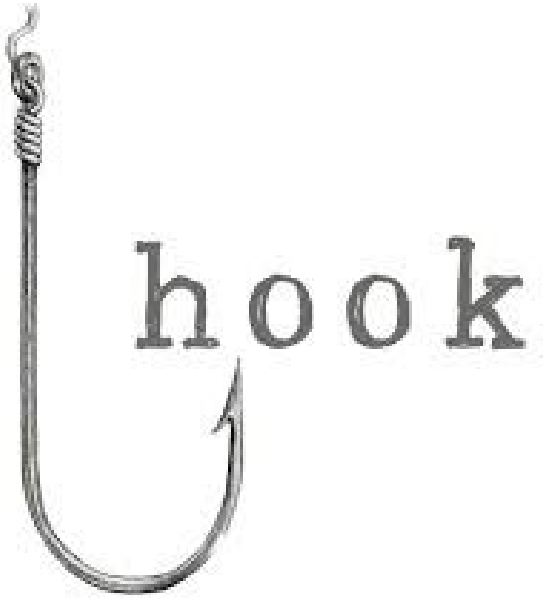
Reglas:

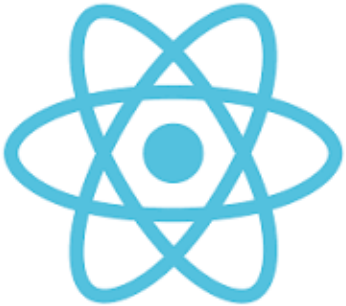
- El value es constante
  - No puedo hacer `name = x`
- Se cambia con `setName`
  - `setName('Nuevo valor')`
- No llamar `setName` entre la declaración del hook y el render

```
import React, { Component, useState } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}

render(<App />, document.getElementById('root'));
```

Reglas generales de los hooks

- Deben ejecutarse **SIEMPRE**
- Esto implica que no pueden ser ejecutados dentro de otras estructuras, como IF, FOR, ó **ternary A ? B : C**
- Se ejecutan en orden y nunca en simultáneo

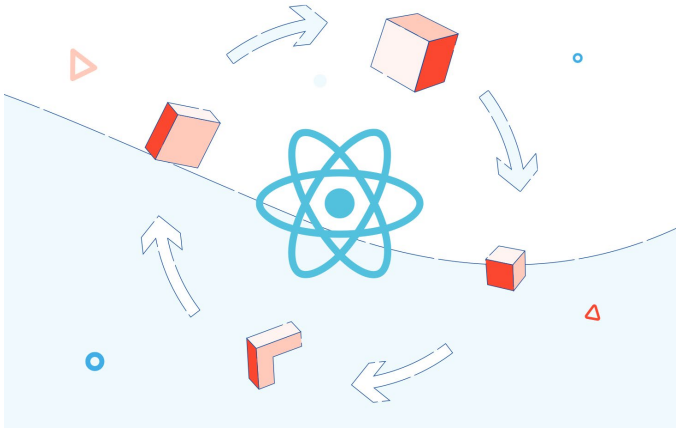




¿Entonces qué correlación hay entre el **render** , las props, el estado y los eventos?

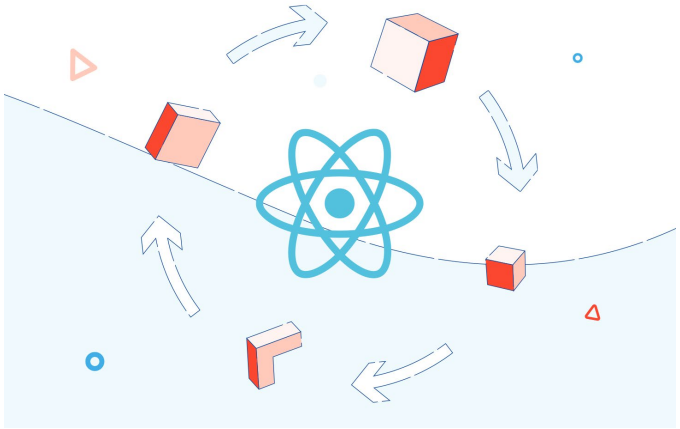
Para saber **qué debe renderizar**, React busca ciertas condiciones específicas:

- Cambio en las **props** `<Title text=”newtext”/>`
- Cambio en el **estado**
  - **`this.setState({count: 2})`** / Class based
  - **`setCount(2)`** / Fn + Hooks
- **Eventos**:
  - Al ocurrir eventos, programáticamente modificaremos el estado, lo cual detona los dos primeros puntos



El ciclo de vida no es más que una **serie de estados** por los cuales **pasa todo componente a lo largo de su existencia** .

Esos estados tienen correspondencia en diversos métodos, que podemos implementar para realizar acciones cuando se van produciendo.



En React es fundamental el ciclo de vida, porque hay determinadas **acciones que necesariamente debemos realizar en el momento correcto de ese ciclo** .

Conocer estos ciclos nos ayudará a optimizar la aplicación, siguiendo las reglas básicas que pone React

Las tres clasificaciones de estados dentro de un ciclo de vida

- El **montaje** se produce la primera vez que un componente va a generarse, incluyéndose en el DOM.
- La **actualización** se produce cuando el componente ya generado se está actualizando.
- El **desmontaje** se produce cuando el componente se elimina del DOM.

useEffect: Ejemplos/Cheatsheet

Si declaro useEffect(() => { //Accion; return **cleanup-fn** })	Si mi acción se ejecuta el montado y **en cada render**, mi limpieza se ejecuta **en cada render.**
Si declaro useEffect(() => { return **cleanup-fn** }, [])	Si mi acción se realiza **al montar**, la limpieza será **únicamente al desmontar** el componente
Si declaro useEffect(() => { return **cleanup-fn** }, [prop])	Mi acción se realizará al montar, y antes del próximo cambio de prop se hará una limpieza y recién ahí se ejecutará la acción

- Toda acción del effect-hook se ejecuta al montar
- Ningún efecto bloquea el render
- Todas las acciones y limpiezas se realizan en orden
- Si modifico el state incluido en los filtros propios habrá un loop infinito

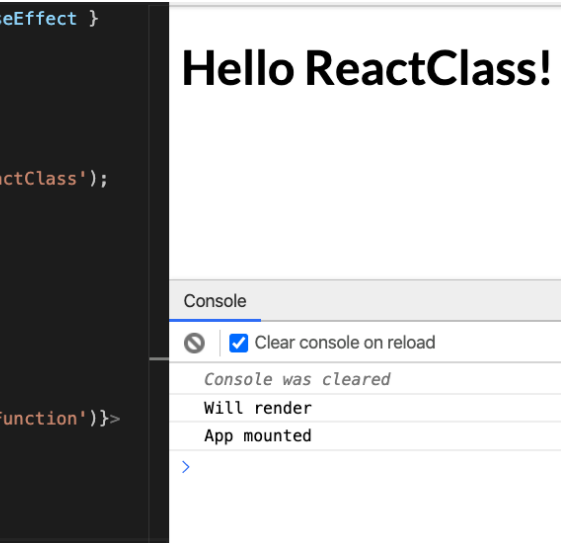
Hook de efecto/useEffect

Podemos utilizar el hook de efecto

```
import React, { Component, useState, useEffect }
from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  useEffect(() => {
    console.log('App mounted');
    return () => {
      console.log('Will unmount');
    }
  }, []);
  console.log('Will render')
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

2

1



useEffect: Cleanup

Si devuelves una función

`return () => {}`

se ejecutará el clean que quieras (ajax call, remover una suscripción, librería, etc)

```
import React, { Component, useState, useEffect }
from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  useEffect(() => {
    console.log('App mounted');
    return () => {
      console.log('Will unmount');
    }
  }, []);
  console.log('Will render')
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

2

1

Hello ReactClass!

Console

Clear console on reload

Console was cleared

Will render

App mounted

IMPORTANTE

Tanto los **callbacks** como los **cleanups** :

- Se ejecutan **en el orden en que se hayan declarado** los otros hooks respectivos
- Recuerda que **la función se destruye en cada ejecución** , si tienes actividad pendiente hay que cerrarla en cada cleanup y volver a suscribirla