



El futuro digital  
es de todos

MinTIC



Vigilada Mineducación

# CICLO IV:

## Desarrollo de Aplicaciones Web

Mision  
TIC2022



El futuro digital  
es de todos

MinTIC

**UN** UNIVERSIDAD  
**DEL NORTE**

Vigilada Mineducación

# Sesión 07:

# Desarrollo de Aplicaciones Web

Desarrollo de Front-End web con React - Patrones de Diseño

Mision  
TIC2022



# Objetivos de la sesión

Al finalizar esta sesión estarás en capacidad de:

1. Implementar patrones de diseño de React en proyectos de desarrollo de aplicaciones web.



# React - Componentes

- Dado que trabajaremos con componentes funcionales hablaremos principalmente de su estructura. Para esto tomemos en cuenta el siguiente componente `<Hola />`:
  - De esto podemos notar que el nombre de nuestro archivo y componente suele seguir la nomenclatura Pascal.
  - Siempre son funciones, por lo general usamos funciones de tipo flecha o arrow functions.
  - En versiones recientes se omite la necesidad de importar React al inicio de nuestros archivos.
  - Se considera una buena práctica el hacerlo para especificar que estamos usando React.

```
import React from 'react';  
  
const Hola = () => <p>Hola!</p>;  
  
export default Hola;
```



# React - Componentes

- Dado que trabajaremos con componentes funcionales hablaremos principalmente de su estructura. Para esto tomemos en cuenta el siguiente componente `<Hola />`:

```
import React from 'react';  
  
const Hola = () => <p>Hola!</p>;  
  
export default Hola;
```



# React - Componentes

- De esto podemos notar que el nombre de nuestro archivo y componente suele seguir la nomenclatura Pascal.
- Siempre son funciones, por lo general usamos funciones de tipo flecha o arrow functions.
- En versiones recientes se omite la necesidad de importar React al inicio de nuestros archivos.
- Se considera una buena práctica el hacerlo para especificar que estamos usando React.



# React - Componentes

- Para utilizar nuestro componente `<Hola />` simplemente lo tenemos que importar en el archivo que lo necesite de la siguiente forma:

```
import Demo from '../ruta/hasta/Hola'
```

- Luego podemos llamar nuestro componente de las siguientes formas:

```
<Hola></Hola>    <Hola />
```

- La segunda opción es un componente que no tiene elementos o nodos hijos, se conoce como self-closing component o componente autocerrado.
- Es considerado una buena práctica utilizar self-closing components tanto como sea posible.



# React - Propiedades

- Ya que nuestros componentes son funciones, nosotros podemos parametrizarlas con un objeto props, nuestros componentes de la siguientes formas:

```
import React from 'react';
```

```
const Mensaje = props =>  
<p>{props.msg}</p>;
```

```
export default Mensaje;
```

```
import React from 'react';
```

```
const Mensaje = ({msg, ...rest}) =>  
<p>{msg}</p>;
```

```
export default Mensaje;
```





# React - Propiedades

- En la segunda forma usamos desestructuración de objetos u object destructuring.
- Como propiedades opcionales definimos el objeto rest para incluir el resto de props de nuestro componente.
- Usar los nombres rest y props es considerado una buena práctica.
- Definimos una prop con el nombre name.



# React - Propiedades

- Para utilizar un componente especificando sus propiedades, o props, podemos proceder de la siguiente forma, seguiremos nuestro ejemplo utilizando <Mensaje>:

```
<Mensaje msg="Hola!" />
```

```
const msg = 'Hola!'
```

```
const props = { msg: 'Hola!' }
```

```
<Mensaje msg = {msg} />
```

```
<Mensaje {...props} />
```

- En el primer caso simplemente pasamos como parámetro una cadena de texto estática.
- En la segunda forma utilizamos interpolación de variables, por lo que en este caso es necesario utilizar llaves para hacer referencia a la variable msg.
- En la tercera forma nuevamente usamos object destructuring.



# React - Estado

- Considere el siguiente componente:

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

const Counter = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>
        +
      </button>
      <button onClick={() => setCount(count - 1)}>
        -
      </button>
    </div>
  );
};

export default Counter;
```



# React - Estado

- Como podemos ver usaremos un método propio de react llamado useState, a este método se le conoce como Hook.
- Este método nos retorna un vector con una variable y un método, es decir [variable, método].
- Esta asignación se puede hacer mediante object destructuring.



# React - Estado

- El método es utilizado para actualizar el valor de nuestra variable.
- Acepta un valor inicial como parámetro.
- Los componentes que manejan estados se conocen como contenedores y los que no se conocen como presentacionales

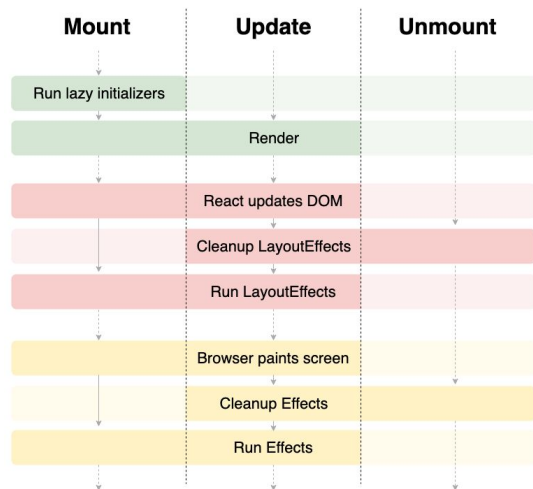


# React - Ciclos de Vida

- Los componentes en react tienen los siguientes ciclos de vida:

## React Hook Flow Diagram

v1.3.1 [github.com/donavon/hook-flow](https://github.com/donavon/hook-flow)



Tomado de [GitHub/@donavon](https://github.com/donavon)

- **Montaje:** Es donde se inicializa nuestro componente.
- **Actualización:** Es donde se actualiza nuestro componente y podemos hacer acciones asíncronas correctamente.
- **Desmontaje:** Es donde se limpian eventos configurados por nosotros y se desmonta y destruye nuestro componente.
- Para ciclos de vida de componentes basados en clases ver este [enlace](#).



# React - Ciclos de Vida

- Para esto React introduce el useEffect Hook.
- Considere el siguiente componente:

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

const Actualizable = () => {
  const [text, setText] = useState('');

  useEffect(() => {
    const espera = setTimeout(() => setText('Ahora tengo texto'), 5000);
    return () => clearTimeout(espera);
  }, []);

  return <p>{text}</p>;
};

export default Actualizable;
```



# React - Ciclos de Vida

- `<Actualizable />` no contiene texto, sino después de cinco segundos desde su montaje.
- Esto sucede porque a través del `useEffect` Hook podemos configurar un proceso asíncrono en el cual actualizamos el estado del componente de estar vacío "", a contener "Ahora tengo texto". Provocando así que `<Actualizable />`, se actualice y re-renderice.
- Esto funciona porque el `useEffect` Hook funciona recibiendo un `callback` y un listado de dependencias.
- Ya que nuestro listado de dependencias le estamos informando a nuestro componente que se actualice en el montaje y desmontaje del mismo.
- El desmontaje ocurre con el return de nuestro `callback`, en este caso solo limpiamos el `timeout`, ya que es un efecto secundario que se eyecturaria si no es removido en el desmontaje.





# React - Ciclos de Vida

- Considere el siguiente componente:

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

const Actualizable = ({ mensaje }) => {
  const [text, setText] = useState('');

  useEffect(() => {
    const espera = setTimeout(() => setText(mensaje), 5000);
    return () => clearTimeout(espera);
  }, [mensaje]);

  return <p>{text}</p>;
};

export default Actualizable;
```



# React - Ciclos de Vida

- `<Actualizable />` ahora recibe props, dentro de las cuales solo acepta prop.
- De igual forma el sigue realizando los mismos procesos que realizaba antes en montaje y desmontaje.
- Las únicas diferencias son:
  - En el montaje, nuestro `timeout` asigna el valor de prop con `setText` a la variable `text`.
  - Nuestro tiene `useEffect` Hook como dependencia a prop.
- Estos cambios hace que `<Actualizable />`, se actualice con cada cambio de la variable `prop` que recibe mediante `props`.
- Lo cual provoca que su contenido se actualice a los cinco segundos desde que recibió cambios.



# React - Manejo de Eventos

- Podemos escuchar eventos propios de los elementos HTML de la siguiente forma:

```
<button onclick="activateLasers()"> Activate Lasers </button>
```

- La traducción directa a React seria:

```
<button onClick={activateLasers}> Activate Lasers </button>
```

- De esto podemos notar que los eventos se escriben en la nomenclatura Camel.
- Adicionalmente todos los eventos disponibles en HTML se pueden consumir desde React.



# React - Manejo de Eventos

Entonces podemos reescribir nuestro component `<Counter />` de la siguiente forma:

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

const Counter = () => {
  const [count, setCount] = useState(0);
  const addHandler = () => setCount(count + 1);
  const subHandler = () => setCount(count - 1);
  const overHandler = () => {console.log(count)};

  return (
    <div onMouseEntered={overHandler}>
      <p>{count}</p>
      <button onClick={addHandler}> + </button>
      <button onClick={subHandler}> - </button>
    </div>
  );
};

export default Counter;
```



# React - Manejo de Eventos

- De esta forma estamos tres eventos que usan el estado de `<Counter />`
  - **addHandler**: agrega uno a nuestro contador.
  - **subHandler**: resta uno a nuestro contador.
  - **overHandler**: imprime nuestro contador en la consola de las herramientas de desarrollador.
- Cabe resaltar que esta forma de definir los eventos es muy estática porque estamos encapsulando nuestros eventos dentro `<Counter />`.



# React - Manejo de Eventos

- Por lo que podemos aceptar eventos personalizados de la siguiente forma:

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

```
const Counter = ({content, onAdd, onSub, onEnter, onLeave}) => {
```

```
  return (  
    <div onMouseEntered={onEnter} onMouseLeave={onLeave}>  
      <p>{content}</p>  
      <button onClick={onAdd}> + </button>  
      <button onClick={onSub}> - </button>  
    </div>  
  );
```

```
};
```

```
export default Counter;
```



# React - Manejo de Eventos

- Finalmente podemos utilizar `<Counter />` de la siguiente forma:

```
import React, { useState } from 'react';
import Counter from './ruta/hasta/Counter';

const Wrapper = () => {
  const [count, setCount] = useState(0);

  const onAdd = () => setCount(count + 1);
  const onSub = () => setCount(count - 1);
  const onEnter = () => { console.log('Entre a <Counter />'); };
  const onLeave = () => { console.log('Sali de <Counter />'); };

  return <Counter content={content} onAdd={onAdd} onSub={onSub} onEnter={onEnter}
onLeave={onLeave} />;
};

export default Wrapper;
```



# React - Manejo de Eventos

- Desde `<Counter />` hemos definido cuatro eventos personalizados:
  - `onAdd.`
  - `onSub.`
  - `onEnter.`
  - `onLeave.`
  - Es considerado una buena práctica seguir la nomenclatura “on” + nombre de la acción a realizar siguiendo la nomenclatura camel.
- De esta forma podemos definir nuestros eventos personalizados cómo lo hacemos en `<Wrapper />` para definir eventos que se ajusten nuestras necesidades.
- Esto nos permite crear componentes reusables como por ejemplo un botón o campo de texto.





# React - Renderizado Condicional

- Finalmente podemos mostrar elementos condicionalmente desde nuestros componentes utilizando el ternary operator u operador ternario de la siguiente forma:

```
{ condicion ? <p>Condición verdadera</p> : <p>Condición falsa</p> }
```

- También podemos usar solo una condición de las siguientes formas:

```
{ condicion ? <p>Condición verdadera</p> : null }
```

```
{ condicion ? null: <p>Condición falsa</p> }
```

- En caso de que solo necesitemos la condición verdadera podríamos hacer lo siguiente:

```
{ condicion && <p>Condición verdadera</p>}
```



# React - Renderizado Condicional

- Por otro lado, podemos definir métodos que se encarguen de este renderizado ya que utilizar operadores ternarios puede causar que nuestro código sea más difícil de entender:

```
const metodo = (/* parametros necesarios */) => {  
  /* Operaciones que definen nuestro contenido con base en nuestros parámetros  
  necesarios, propiedades o el estado de nuestro componente */  
  
  const contenido = 'Algún contenido'  
  return contenido  
}  
  
{ metodo() } /* Esta sintaxis sólo puede usar dentro de JSX  
  
puesto que es la forma de hacer interpolaciones */
```



# React - Renderizado Condicional

- Esto es posible ya que JSX nos permite interpolar una expresión, ya sea un método, una variable o una operación ternaria.
- Por último, podemos reescribir `<Mensaje />` para renderice su mensaje de forma condicional:

```
import React from 'react';

const Mensaje = ({ show , msg, ...rest }) => {
  return show && (<p>{msg}</p>)}
}

export default Mensaje;
```

- De esta forma mostrar la prop `msg` depende de la prop `show`.



El futuro digital  
es de todos

MinTIC

**UN** UNIVERSIDAD  
**DEL NORTE**

Vigilada Mineducación

# Ejercicios de práctica

Mision  
TIC2022



# Referencias

- <https://reactjs.org/docs/components-and-props.html#gatsby-focus-wrapper>
- <https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841>
- <https://reactjs.org/docs/handling-events.html>
- <https://reactjs.org/docs/conditional-rendering.html>



El futuro digital  
es de todos

MinTIC

**UN** UNIVERSIDAD  
**DEL NORTE**

Vigilada Mineducación

**¡GRACIAS**  
**POR SER PARTE DE**  
**ESTA EXPERIENCIA**  
**DE APRENDIZAJE!**



**Misión**  
**TIC 2022**