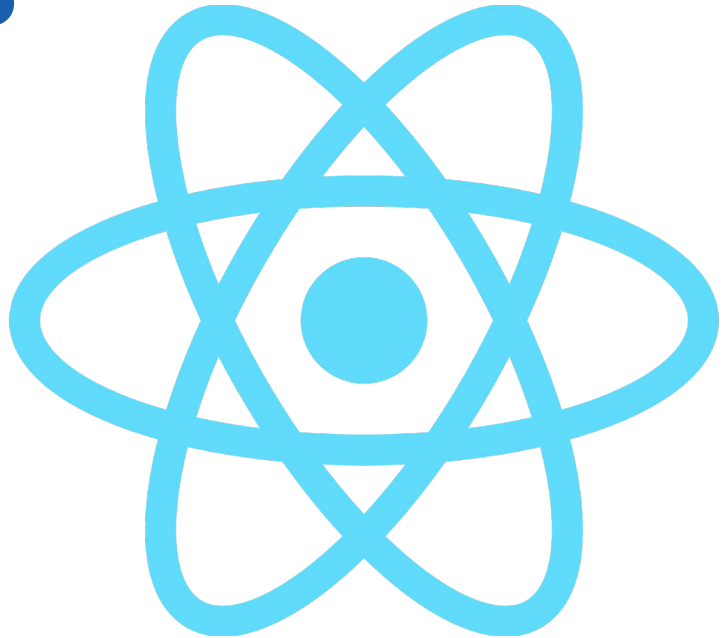
Clase 3

REACT



Temario

1. React
2. React Components and Props
3. **Composing React Components and Stateful React Components**
4. Events, conditional rendering
5. Lists and forms
6. React Router I
7. React Router II
8. Debugging and Testing

Composición de Componentes

La composición de componentes es un poderoso patrón para hacer los componentes más reutilizables.

En React, podemos hacer que los componentes sean más genéricos aceptando props, que son para los componentes React lo que los parámetros son para las funciones.

Composición de componentes es el nombre para pasar componentes como props a otros componentes, creando así nuevos componentes con otros componentes.

Existe una prop especial llamada ***children*** muy recomendable para pasar elementos hijos directamente en su resultado.

```
const Button = (props) => {
  <button onClick={props.onClick}>{props.children}</button>
};
```

```
const Button = (props) => {
  const { onClick, children } = props;
  return <button onClick={onClick}>{children}</button>;
};
```

```
const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);

const App = () => {
  const onClick = () => alert("Hey 👋");

  return <Button onClick={onClick}>Click me!</Button>;
};
```

Composición de Componentes

Algunos componentes no conocen a sus hijos de antemano, por ejemplo componentes como Sidebar o Dialog que representan “cajas” genéricas.

A través de children se pueden pasar elementos hijos directamente a un componente para usarlos en su resultado. De esta manera podremos reutilizar nuestros componentes con diferentes propósitos. Por ejemplo:

- Vamos a crear una app my-sixth-app
- Dentro de App.js creamos un componente FancyBorder y dos componentes WelcomeDialog y GoodByeDialog que lo utilice
- El componente WelcomeDialog debe renderizar el componente FancyBorder con un título h1 “Bienvenido!” y un p con el texto ”Explora todos nuestros productos!” como hijos
- Además, el componente GoodByeDialog debe renderizar el componente FancyBorder con un título h1 “Adiós!” y un p con el texto ”Gracias por visitar nuestra web!” como hijos.
- El componente App deber renderizar un componente WelcomeDialog y GoodByeDialog

```
// src/App.js

const FancyBorder = (props) => {
  return (
    <div className={"FancyBorder FancyBorder-" + props.color}>
      {props.children}
    </div>
  );
};

const WelcomeDialog = () => {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title"> Bienvenido! </h1>
      <p className="Dialog-message"> Explora todos nuestros productos! </p>
    </FancyBorder>
  );
};

const GoodByeDialog = () => {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title"> Adiós! </h1>
      <p className="Dialog-message"> Gracias por visitar nuestra web! </p>
    </FancyBorder>
  );
};

const App = () => {
  return (
    <>
      <WelcomeDialog />
      <GoodByeDialog />
    </>
  );
};

export default App;
```

Prop Drilling o Perforación de Props

En el siguiente ejemplo, tenemos 3 capas, la primera (**App**) envía el userName a la segunda (**WelcomePage**) que a su vez envía la misma prop a la siguiente capa (**WelcomeMessage**).

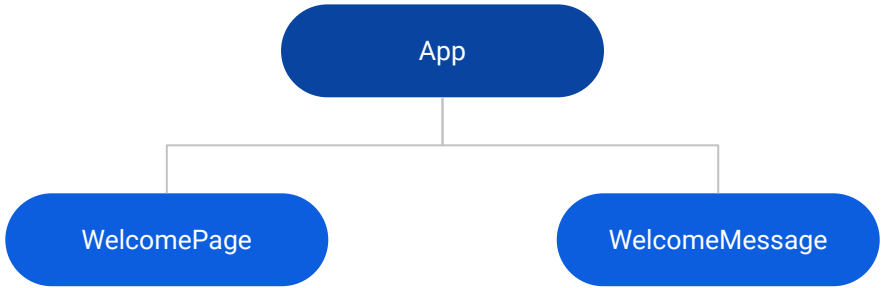
Por tanto tenemos la siguiente estructura:


```jsx
const App = () => {
  const userName = "Joe";

  return <WelcomePage userName={userName} />;
};

const WelcomePage = ({ userName }) => {
  return (
    <>
      <WelcomeMessage userName={userName} />
      {/** Some other welcome page code */}
    </>
  );
};

const WelcomeMessage = ({ userName }) => {
  return <h1>Hey, {userName}!</h1>;
};
```

Prop Drilling o Perforación de Props

Con pocas capas esto no es un gran problema, sin embargo, en aplicaciones más grandes no es práctico ni eficiente. Gracias a la Composición de Componentes podemos darle solución a esta problemática.

Al eliminar una capa tenemos la siguiente estructura:




```jsx
const App = () => {
  const userName = "Joe";

  return <WelcomePage title={<WelcomeMessage userName={userName} />} />;
};

const WelcomePage = ({ title }) => {
  return (
    <>
      {title}
      {/** Some other welcome page code */}
    </>
  );
};

const WelcomeMessage = ({ userName }) => {
  return <h1>Hey, {userName}!</h1>;
};
```

Performance

El patrón de Composición de Componentes es muy recomendable para reducir el número de renderizados en nuestra aplicación.

Por poner un ejemplo, si tenemos un componente Post que muestra el progreso del scroll, React actualiza el componente en base al evento de scroll.

En nuestro ejemplo, si el contenido del Post tiene muchos componentes, el renderizado cada vez que cambia el scroll sería muy costoso.

Esto se podría solucionar separando la lógica del progreso a otro componente y utilizando la composición de componentes.

```
const Post = () => {
  const [progress, setProgress] = React.useState(0);
  React.useEffect(() => {
    const scrollListener = () => {
      // update the progress based on the scroll position
    }

    window.addEventListener('scroll', scrollListener, false);
  }, [])

  return (
    <>
      <h2 className="progress">
        Progress: {progress}%
      </h2>
      <div className="content">
        <h1>Content Title</h1>
        {/** more content */}
      </div>
    </>
  )
}
```

Performance

Si refactorizamos nuestro código anterior y separamos la lógica para pasar el contenido del Post como hijo (children), conseguimos que no se renderice ya que React renderiza props sólo si han cambiado.

Podéis ver el ejemplo funcional aquí:

https://codepen.io/fgerschau/pen/QWaRgKg

```
const PostLayout = ({ children }) => {
  const [progress, setProgress] = React.useState(0);
  React.useEffect(() => {
    const scrollListener = () => {
      // update the progress based on the scroll position
    };

    window.addEventListener("scroll", scrollListener, false);
  }, []);

  return (
    <>
      <h2 className="progress">Progress: {progress}%</h2>
      {children}
    </>
  );
};

const Post = () => {
  return (
    <PostLayout>
      <div className="content">
        <h1>Content Title</h1>
        {/** more content */}
      </div>
    </PostLayout>
  );
};
```

Estado

En nuestra aplicación my-fifth-app estamos almacenando los datos de las personas en un array en una variable, y pasándolos como props. Esto está bien para empezar, pero imagina si queremos ser capaces de eliminar un elemento de la matriz. Con props, tenemos un flujo de datos unidireccional, pero con state podemos actualizar datos privados desde un componente.

Puedes pensar en el estado como cualquier dato que deba ser guardado y modificado sin ser necesariamente añadido a una base de datos - por ejemplo, añadir y eliminar artículos de un carrito de la compra antes de confirmar la compra.

Estado: Ejemplo

Para comprobar el funcionamiento de los estados vamos a añadir funcionalidad a nuestra tabla de la app my-fifth-app de la clase 2 para poder eliminar personas de la tabla. Para ello:

- Copiamos la app my-fifth-app en my-seventh-app
- En la clase App dentro del archivo App.js creamos un objeto state para almacenar las personas. Ahora podemos pasarle estos datos a la tabla mediante this.state.people
- Creamos un nuevo método llamado removePeople que reciba el índice como argumento, a través de un método filter debería eliminar la persona del índice recibido y setearlo al estado usado **setState**
- Pasamos las personas y la función removePeople como props
- Añadimos el botón para eliminar una persona en el componente **TableBody**

```
// src/App.js
import React, { Component } from "react";
import Table from "./Table";
class App extends Component {
  state = {
    people: [
      {
        name: "John",
        job: "Developer",
      },
      {
        name: "Maya",
        job: "Architect",
      },
    ],
  };
  removePeople = (index) => {
    const { people } = this.state;
    this.setState({
      people: people.filter((character, i) => {
        return i !== index;
      }),
    });
  };
  render() {
    const title = <h1>Nice People</h1>;
    return (
      <div className="container">
        <Table
          peopleData={this.state.people}
          removePeople={this.removePeople}
          title={title}
        />
      </div>
    );
  }
}

export default App;
```

Estado: Ejemplo

- En el componente Table recuperamos la prop removePeople y la pasamos a TableBody
- Añadimos el botón para eliminar una persona en el componente **TableBody** usando la prop removePeople

```
// src/Table.js
import React, { Component } from "react";

const TableHeader = () => {
  return (
    <thead>
      <tr>
        <th>Name</th>
        <th>Job</th>
        <th>Remove</th>
      </tr>
    </thead>
  );
};

function TableBody(props) {
  const rows = props.peopleData.map((row, index) => {
    return (
      <tr key={index}>
        <td>{row.name}</td>
        <td>{row.job}</td>
        <button onClick={() => props.removePeople(index)}>Delete</button>
      </tr>
    );
  });

  return <tbody>{rows}</tbody>;
}

class Table extends Component {
  render() {
    const { peopleData, removePeople, title } = this.props;
    return (
      <>
        {title}
        <table>
          <TableHeader />
          <TableBody peopleData={peopleData} removePeople={removePeople} />
        </table>
      </>
    );
  }
}

export default Table;
```

Estado: Ejemplo

Hasta ahora hemos añadido la funcionalidad para eliminar una persona, sin embargo, es habitual en una aplicación añadir registros dinámicamente, por tanto, vamos a añadir un formulario para hacerlo y actualizar nuestro estado.

- Eliminamos todos los datos hardcodeados de people. Lo inicializamos como un array vacío.
- Creamos un nuevo componente Form:
  - Creamos un estado inicial
  - Añadimos una función para actualizar el estado con el valor de los campos del formulario
  - Creamos el formulario con un campo name y job y un botón de Submit
  - Creamos una función submitForm que enviará los datos al prop handleSubmit

```
// src/Form.js
import React, { Component } from "react";

class Form extends Component {
  initialState = {
    name: "",
    job: "",
  };
  state = this.initialState;
  handleChange = (event) => {
    const { name, value } = event.target;

    this.setState({
      [name]: value,
    });
  };
  submitForm = () => {
    this.props.handleSubmit(this.state);
    this.setState(this.initialState);
  };
  render() {
    const { name, job } = this.state;
    return (
      <form>
        <label htmlFor="name">Name</label>
        <input
          type="text"
          name="name"
          id="name"
          value={name}
          onChange={this.handleChange}
        />
        <label htmlFor="job">Job</label>
        <input
          type="text"
          name="job"
          id="job"
          value={job}
          onChange={this.handleChange}
        />
        <input type="button" value="Submit" onClick={this.submitForm} />
      </form>
    );
  }
}

export default Form;
```

Estado: Ejemplo

- Igual que con removePeople, creamos una función handleSubmit para añadir una persona en App.js

- Renderizamos debajo de la tabla el componente Form con el prop handleSubmit para enviarle la función que agrega la persona a nuestro estado people.

```
// src/App.js
import React, { Component } from "react";
import Table from "./Table";
import Form from "./Form";

class App extends Component {
  state = {
    people: [],
  };

  removePeople = (index) => {
    const { people } = this.state;
    this.setState({
      people: people.filter((character, i) => {
        return i !== index;
      }),
    });
  };

  handleSubmit = (character) => {
    this.setState({ people: [...this.state.people, character] });
  };

  render() {
    const title = <h1>Nice People</h1>;
    return (
      <div className="container">
        <Table
          peopleData={this.state.people}
          removePeople={this.removePeople}
          title={title}
        />
        <Form handleSubmit={this.handleSubmit} />
      </div>
    );
  }
}

export default App;
```

Estado: Componentes funcionales

Hasta ahora hemos visto cómo se usa un estado en componentes de clase, sin embargo, gracias a los hooks de estado añadidos en React 16.8, podemos y usar estado y otras características de React sin escribir una clase.

- Copiamos nuestra app my-seventh-app en my-seventh-functional-app
- Convertimos nuestra clase App en una función App
- Convertimos las funciones removePeople y handleSubmit en constantes
- Usamos el método useState para manipular el estado e inicializamos con el array de personas vacío
- Usamos la función setPeople para modificar el estado de people
- Cambiamos this.state.people por people, this.state.removePeople por removePeople, this.title por title y this.handleSubmit por handleSubmit

```
// src/App.js
import React, { useState } from 'react';
import Table from "./Table";
import Form from "./Form";

const App = () => {
    const [people, setPeople] = useState([]);
    const removePeople = (index) => {
        setPeople(people.filter((character, i) => {
            return i !== index;
        }));
    };
    const handleSubmit = (character) => {
        setPeople([...people, character]);
    };
    const title = <h1>Nice People</h1>;
    return (
        <div className="container">
            <Table
                peopleData={people}
                removePeople={removePeople}
                title={title}
            />
            <Form handleSubmit={handleSubmit} />
        </div>
    );
};

export default App;
```

Estado: Uso

Para hacer un uso correcto del estado hay que saber 4 cosas:

1. No se puede modificar el estado directamente, hay que utilizar **setState**

```
// Incorrecto
this.state.people = [];
// Correcto
this.setState({ people: [] });
```

2. Las actualizaciones del estado pueden ser asíncronas. Por ejemplo, debido a que this.props y this.state pueden actualizarse de forma asincrónica, no debes confiar en sus valores para calcular el siguiente estado. Para solucionarlo hay que utilizar la segunda forma de setState que acepta una función en lugar de un objeto. Esa función recibirá el estado previo como primer argumento, y las props en el momento en que se aplica la actualización como segundo argumento:

```
// Incorrecto
this.setState({
  counter: this.state.counter + this.props.increment,
});

// Correcto
this.setState((state, props) => ({
  counter: state.counter + props.increment,
}));
```

Estado: Uso

3. Las actualizaciones de estado se fusionan. Cuando invocas a setState(), React combina el objeto que proporcionaste con el estado actual.
   Por ejemplo, tu estado puede contener varias variables independientes (people, posts y comments).

   La función **handlePosts** deja intacto this.state.people y this.state.comments, pero reemplaza completamente **this.state.posts**. Ocurre lo mismo con la función **handleComments**, únicamente reemplaza **this.state.comments**.

4. Los datos fluyen hacia abajo. Ni los componentes padres o hijos pueden saber si un determinado componente tiene o no tiene estado y no les debería importar si se define como una función o una clase. Por eso es que el estado a menudo se le denomina local o encapsulado. No es accesible desde otro componente excepto de aquel que lo posee y lo asigna.

```
state = {
  people: [],
  posts: [],
  comments: [],
};

handlePosts = () => {
  this.setState({
    posts: [{ id: 1, name: "post 1" }],
  });
};

handleComments = () => {
  this.setState({
    comments: [{ id: 1, body: "Comment 1" }],
  });
};
```

Ciclo de vida de un componente

En aplicaciones con muchos componentes, es muy importante liberar recursos tomados por los componentes cuando se destruyen.

Podemos declarar métodos especiales en la clase del componente para ejecutar algún código cuando un componente se monta (***componentDidMount()***) y desmonta (***componentWillUnmount()***):

El método **componentDidMount()** se ejecuta después que la salida del componente ha sido renderizada en el DOM.

El método **componentWillUnmount()** se ejecuta cuando el componente se elimina del DOM.

Ciclo de vida de un componente: Ejemplo

Vamos a crear un ejemplo que haga uso de los métodos de ciclo de vida, para ello:

- Creamos una app my-eighth-app
- En el archivo App.js creamos un componente Clock que:
  - Tenga un estado date con la fecha actual *"new Date();"* como valor inicial
  - Tenga un método tick() que cambie el estado con la fecha actual "new Date;"
  - Al montar el componente se inicialice un intervalo cada 1000ms que ejecute la función tick
  - Al desmontar el componente se debe eliminar el intervalo del punto anterior
  - Debe renderizar un texto que muestre el estado date.
- El componente App debe renderizar el componente Clock

```
// src/App.js
import React from "react";

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }

  componentDidMount() {
    this.timerID = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({ date: new Date() });
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const App = () => {
  return <Clock />;
};

export default App;
```

Ciclo de vida de un componente:
Componentes funcionales

Al igual que con el estado en los componentes de clase, los métodos de ciclo de vida también se puedan usar en componentes funcionales gracias a los hooks.

Existe un hook llamado ***useEffect*** que equivale a componentDidMount, componentDidUpdate y componentWillUnmount combinados. Veámoslo con un ejemplo:

- Copiamos nuestra app my-eighth-app en my-eighth-functional-app
- Convertimos el componente de clase Clock en un componente funcional
- Usamos useState y useEffect para conseguir el mismo efecto

```
// src/App.js
import React, { useState, useEffect } from "react";

const Clock = () => {
  const [date, setDate] = useState(new Date());

  const tick = () => {
    setDate(new Date());
  };

  useEffect(() => {
    const timerID = setInterval(() => tick(), 1000);
    return function cleanup() {
      clearInterval(timerID);
    };
  });

  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {date.toLocaleTimeString()}.</h2>
    </div>
  );
};

const App = () => {
  return <Clock />;
};

export default App;
```

Ejercicios

1. Replicar el código de clase:
    - App my-sixth-app con el código de la sección **Composición de Componentes**
    - App my-prop-drilling-app con el ejemplo para evitar Prop Drilling
    - App my-seventh-app con la app con gestión de estado
    - App my-seventh-functional-app con gestión de estado en componentes funcionales
    - App my-eighth-app con el temporizador usando métodos del ciclo de vida
    - App my-eighth-functional-app usando métodos del ciclo de vida en componentes funcionales
2. Leer la teoría de Composición vs Herencia.

ALBAÑILESDIGITALES

Developing your new career

servicio navarro de empleo

das·nano

VeriDas