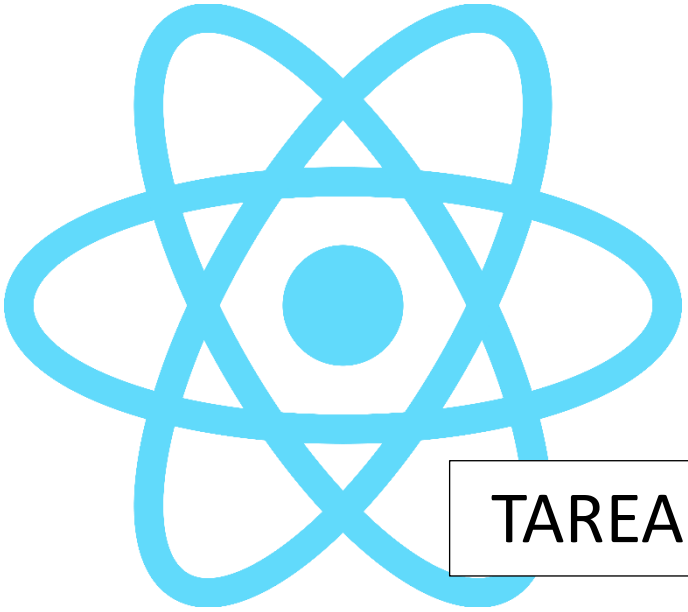


TAREA 3: REACT

Contenido

Parte A: JSX básico (guardar y devolver un elemento)

Edita src/App.jsx para crear un elemento JSX en una variable y devolverlo:

```jsx
function App() {
  const myElement = <h1>Bienvenido al multiverso JSX de Klara</h1>
  return myElement
}
export default App
```

Ilustración 1. Parte A - Guardar y devolver un elemento. Elaboración propia.

En este ejemplo, guardamos una etiqueta JSX dentro de una **variable** para luego devolverla en el return. Esto sirve para demostrar que en React la interfaz se puede manejar como si fuera cualquier otro dato de JavaScript.

Ilustración 2. Resultado en pantalla. Elaboración propia.

Una vez levantado el servidor de Vite y entrado a la dirección que te genera nos mostrara el resultado de la función.

Parte B: Expresiones dentro de JSX con { }

Crea variables y úsalas dentro de JSX. Recuerda: class en HTML es className en React.

Crea una clase .text-lowercase en src/index.css para que se vea el efecto.

```jsx
function App() {
  const lowercaseClass = 'text-lowercase'
  const text = 'Hola, humano esclavizado de DAW2. React detectado check'
return <h1 className={lowercaseClass}>{text}</h1>
}
export default App
```

Ilustración 3. Parte B - Expresiones con {}. Elaboración propia

En este ejemplo, usamos **variables y llaves { }** para pasar datos dinámicos al HTML, como el texto de un título o el nombre de una clase CSS. Esto nos permite separar el contenido de la estructura y hacer que la interfaz cambie según los valores de nuestro código JavaScript.

```css
.text-lowercase {
  text-transform: lowercase;
  color: #4a90e2;
  font-family: sans-serif;
}
```

Ilustración 4. CSS className.
Elaboración propia

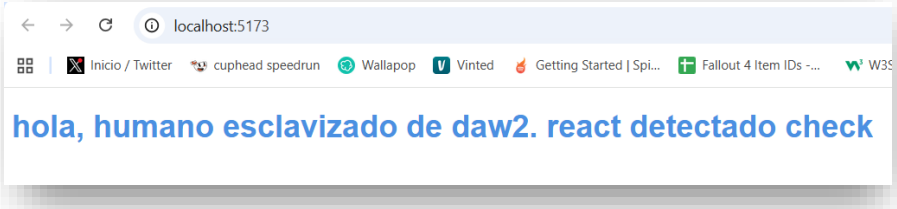
El estilo CSS de la clase de la función creada anteriormente className.

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

Importamos el CSS para que se aplique el estilo a la className

Ilustración 5. Importar CSS. Elaboración propia

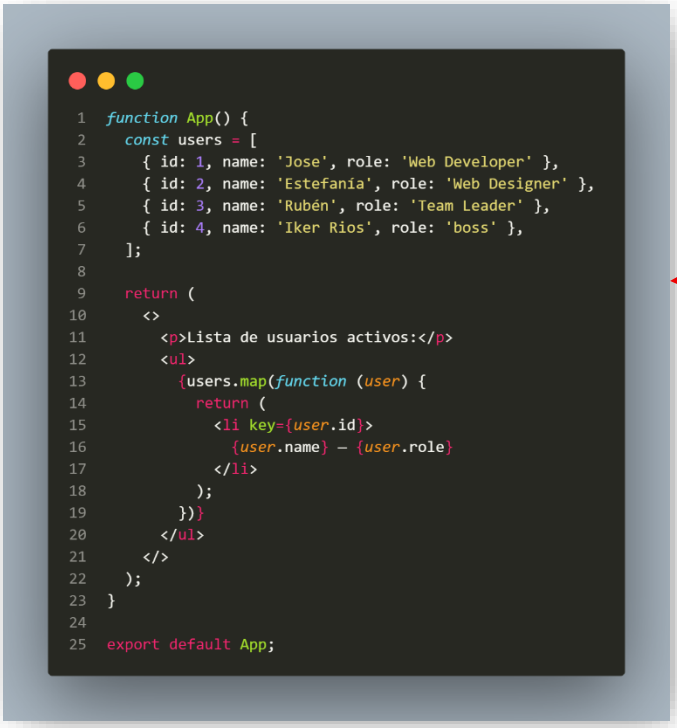


Resultado del estilo aplicado.

Ilustración 6. Visualización del estilo CSS. Elaboración propia

Parte C: Renderizar una lista con map() + key

Además, añade algún usuario más, haciendo uso del nombre de algún compañero de clase.



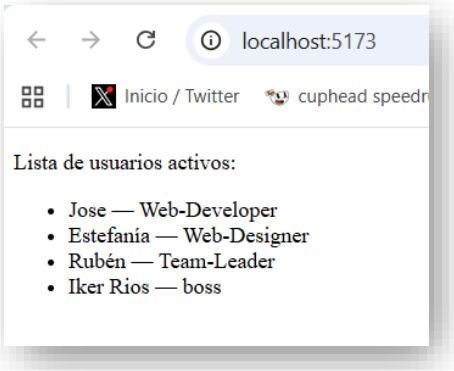
```
function App() {
  const users = [
    { id: 1, name: 'Jose', role: 'Web Developer' },
    { id: 2, name: 'Estefanía', role: 'Web Designer' },
    { id: 3, name: 'Rubén', role: 'Team Leader' },
    { id: 4, name: 'Iker Rios', role: 'boss' },
  ];

  return (
    <>
      <p>Lista de usuarios activos:</p>
      <ul>
        {users.map(function (user) {
          return (
            <li key={user.id}>
              {user.name} – {user.role}
            </li>
          );
        })}
      </ul>
    </>
  );
}

export default App;
```

En este código usamos **.map()** para recorrer una lista de usuarios y dibujarlos todos automáticamente en la pantalla. Usamos **key** para darle a cada uno un "DNI" único y que React no se mude de locos al actualizar la lista.

Ilustración 7. Lista con map() + key . Elaboración propia



Resultado en pantalla de la lista.

Ilustración 8. Visualización del resultado. Elaboración propia

Parte D: className y estilos rápidos

Aplica className a la lista para diferenciar roles (mínimo 2 estilos).

Crea las clases en index.css (o App.css) y verifica el resultado en pantalla.

```jsx
function App() {
  const users = [
    { id: 1, name: 'Jose', role: 'Web-Developer' },
    { id: 2, name: 'Estefanía', role: 'Web-Designer' },
    { id: 3, name: 'Rubén', role: 'Team-Leader' },
    { id: 4, name: 'Iker Rios', role: 'boss' },
  ];

  return (
    <>
      <p>Lista de usuarios activos:</p>
      <ul>
        {users.map((user) => (
          <li key={user.id} className={user.role}>
            <strong>{user.name}</strong> — {user.role}
          </li>
        ))}
      </ul>
    </>
  );
}
```

Aquí recorremos la lista de usuarios con .map() para que se pinten todos solos en la pantalla. Usamos el key para que React no se mude de locos al identificarlos y el className para ponerle un estilo distinto a cada uno según su rol.

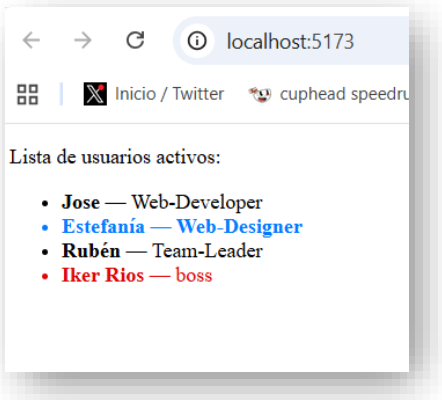
Ilustración 9. className. Elaboración propia

```css
.Web-Designer {
  color: #007bff;
  font-weight: bold;
}

.boss {
  color: #e00404;
}
```

Estilos CSS que se aplicaran según su rol.

Ilustración 10. Estilos CSS de los roles. Elaboración propia



Resultado en pantalla de los estilos CSS aplicados.

Ilustración 11. Visualización de los estilos CSS. Elaboración propia

Preguntas extra para el PDF

1. ¿Qué es JSX con tus palabras?

   Es como escribir HTML pero directamente dentro de JavaScript. Nos sirve para diseñar la interfaz de forma más fácil y rápida en el mismo lugar donde está la lógica.

2. ¿Por qué usamos { } dentro de JSX? Pon un ejemplo.

   Usamos las llaves **{ }** para "escapar" del HTML y meter código de JavaScript directamente. Sirven para mostrar valores dinámicos, como variables, operaciones matemáticas o el resultado de una función.

   ```
   const nombre = "Pepe";
   <h1>Hola, {nombre}</h1> // Se verá: Hola, Pepe
   ```

3. ¿Para qué sirve la prop key en listas?

   La prop **key** sirve para darle una identidad única a cada elemento de una lista. Así, cuando algo cambia (se borra o se edita un ítem), React sabe exactamente qué pieza tocar sin tener que volver a renderizar toda la lista desde cero.

4. ¿Por qué usamos className y no class en React?

   Usamos **className** porque React utiliza JavaScript para definir la interfaz, y en JavaScript la palabra class ya está reservada para crear clases de objetos.