ACTIVITAT AVALUABLE AC1
Mòdul: MP06- Desenvolupament web en entorn client **UF:** UF3: Esdeveniments. Manejament de formularis. Model d'objectes del document **Professor:** Albert Guardiola **Data límit d'entrega:** 3/3/2025 23:59 **Mètode d'entrega:** Per mitjà del Clickedu de l'assignatura. Les activitats entregades més enllà de la data límit només podran obtenir una nota de 5. **Instruccions:** S'ha d'entregar un únic document amb el nom: ***MP06-UF3-AC1-Nom_Alumne.pdf*** Es valorarà la presentació.

Introducció a la llibreria React

1. Desplega aquest projecte amb JS. Fixa't que la funció renderApp s'encarrega de carregar en el DOM (a l'element #app) un contingut HTML. L'HTML, a part d'aquest contenidor, és buit.

Veurem que aquesta és una de les idees centrals de React: **els components del frontend es declaren i renderitzen des de JS**.

```
<html>
  <body>
    <div id="app"></div>
    <script src="./script.js"></script>
  </body>
</html>
```

```
window.onload = () => {
    renderApp();
}

function renderApp() {
    let app = document.getElementById('app');
    let header = document.createElement('h1');
    let text = 'Develop. Preview. Ship.';
    let headerContent = document.createTextNode(text);
    header.appendChild(headerContent);
    app.appendChild(header);
}
```

2. Reproduirem ara el mateix frontend, fent servir la llibreria React. La podríem descarregar i afegir a les dependències del nostre projecte, però serà més còmode carregar-les des d'un CDN remot:

```
<script src="https://unpkg.com/react@18/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

L'script farà servir l'**objecte ReactDOM** per a renderitzar el contingut HTML.

```
function renderApp() {
    const app = document.getElementById('app');
    const root = ReactDOM.createRoot(app);
    root.render(<h1>Develop. Preview. Ship.</h1>);
}
```

Si executem l'aplicació, veureu que la consola traça el següent **error de sintaxis**:

```
Terminal
Uncaught SyntaxError: expected expression, got '<'
```

➔ **Per què ocorre l'error?**

Efectivament, en el codi JS hi ha un fragment (concretament, la cadena que passem al *root.render*, que no és JS vàlid.

De fet, aquest fragment està escrit en JSX, una extensió de JS que permet escriure estructures de DOM de manera senzilla, directament en el codi JS.

➔ **A quines línies del cdi en vanilla JS substitueix el JSX <h1>Develop. Preview. Ship.</h1>)?**

Fixa't també que el *root.render* pren habitualment (gairebé sempre, a la pràctica) un fragment JSX com entrada.

3. Ocupem-nos ara de l'error de sintaxis. El JSX s'ha de compilar per traduir-lo a JS vàlid. Aquesta compilació la farà la llibreria Babel, que també afegirem al nostre projecte via CDN:

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

Per fer que Babel compili el nostre script abans que s'executi, hem de canviar el tipus d'script a

```
type='text/jsx'    o    type='text/babel
```

Fets els canvis, torna a carregar l'aplicació al navegador i observa com el frontend es desplega correctament.

➔ **Observa també com la consola del navegador informa que s'està transformant el codi JSX en JS vàlid.**

4. Llenguatge imperatiu vs llenguatge declaratiu:

➔ **Donat el que hem vist fins ara, pensa per què es diu que (vanilla) JS és un llenguatge imperatiu mentre que JSX és un llenguatge declaratiu.**

➔ **Posa altres exemples de llenguatge imperatiu i declaratiu, respectivament.**

5. A React, l'HTML que es renderitza s'agrupa en components.

Un component de React es declara com una funció. Refactoritza el codi de l'app per que el *root.render* carregui el component *header.*

```
function renderApp() {
    const app = document.getElementById('app');

    function header(){
        return (<h1>Develop. Preview. Ship.</h1>);
    }

    const root = ReactDOM.createRoot(app);
    root.render(header);
}
```

Observa la traça d'avís en carregar de nou l'aplicació:

Warning: Functions are not valid as a React child. This may happen if you return a Component instead of <Component /> from render. Or maybe you meant to call this function rather than return it.

La llibreria React exigeix que:
1. els components estiguin declarats amb majúscula inicial
2. i que es passin al *root.render* amb notació d'etiqueta HTML.

```
function renderApp() {
    const app = document.getElementById('app');

    function Header(){
        return (<h1>Develop. Preview. Ship.</h1>);
    }
```

```
    const root = ReactDOM.createRoot(app);
    root.render(<Header />);
}
```

Fes els canvis al codi, recarrega l'aplicació i observa com el component Header es desplega correctament.

➔ **Observa que un component de React és:**
1. **Una funció…**
2. **… que retorna un fragment… escrit en quin llenguatge?**

DISCLAIMER: Aquells de vosaltres que ja hagueu fet servir React, en el 90% dels casos l'haureu fet servir a dins d'un *framework* o *bundler* més complex: *NextJS, Vite…* I és així com s'acostuma a fer servir habitualment:
1. Per facilitar el desplegament.
2. Per integrar components de servidor (inclús els propis de React, com ReactSever, per exemple)

En breu, nosaltres passarem a fer servir un *framework full stack*. Però de mment, estem estudiant la llibreria React "pelada" per insistir en els seus conceptes fonamentals. Paciència, que tot arriba.

El que pot sobtar a una persona que ha conegut react en el context d'un *framework* més complet és que no estiguem declarant els components en mòduls JS separats. Bé, això s'explica per una incompatibilitat amb JSX: JSX no permet mòduls de manera nativa.

6. React només pot renderitzar un component *root*. La resta de components han d'estar aniuats a dins seu.

Refactoritza el codi perquè es carregui el component *HomePage*, que alhora inclou en el seu JSX el component *Header*.

```
// Componente HomePage
    function HomePage() {
        return <div></div>;
    }
```

```
// Componente HomePage
    function HomePage() {
        return (
            <div>
                <Header />
```

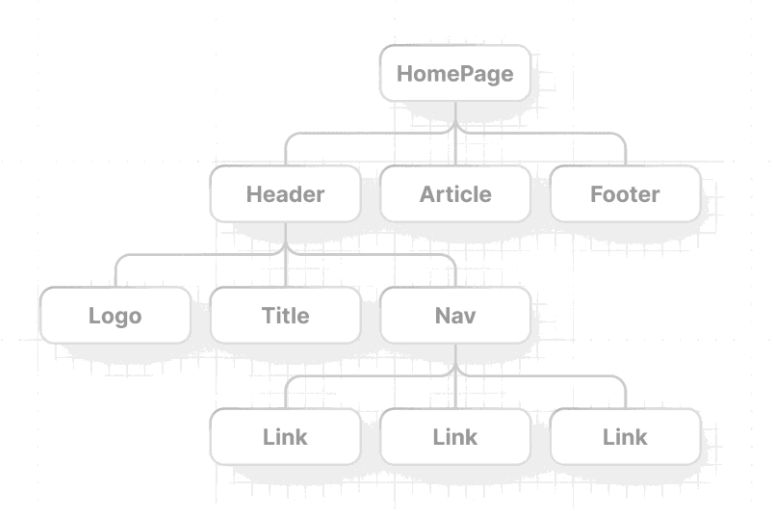
```
            </div>
        );
    }

root.render(<HomePage />);
```

➔ **Per tant, quants components pot carregar el *root.render*?**

7. Aquesta "limitació" fa que els frontends creats amb React hagin de tenir estructura d'arbre.

➔ **Pensa com carregaries el següent arbre de components en React. No es demana implementar-lo al codi que estem fent servir d'exemple!**



8. Seguim amb el codi de la nostra HomePage.

El component pare (en aquest cas HomePage), pot voler decidir certes propietats o valors del component fill (Header). Això és possible declarant **props** en el component fill.

Modifica el codi per tal que:

1. El component fill tingui un objecte **props** com entrada, i que pinti la propietat *title* contingut de l'h1.
2. El component pare *passi* al fill el valor "Hola Mundo" per a la propietat *title*.

```
function Header(props) {
        return <h1>{props.title}</h1>;
    }
```

```
function HomePage() {
        return (
            <div>
                <Header title="Hola Mundo" />.
            </div>
        );
    }
```

➔ **De quina paraula és abreviatura *prop*?**

Per un ús més còmode de les *props*, podem fer servir la sintaxi de desestructuració d'objectes de JS. Observa:

Component pare (no canvia):

```
 function HomePage() {
        return (
            <div>
                <Header title="Bienvenido" user="Albert"/>
            </div>
        );
    }
```

Component fill (sense desestructuració):

```
function Header(props) {
        return <h1>{props.title} {props.user}</h1>;
    }
```

Component fill (amb desestructuració):

```
function Header({title,user}) {
        return <h1>{title} {user}</h1>;
    }
```

➔ **Pot un component fill canviar el valor de les propietats que li passa el pare? (es diu que les *props* són immutables).**

➔ **L'anterior no impedeix que es puguin definir (en el fill) valors per defecte de les *props*. Investiga quina és la sintaxi per fer-ho.**

9. Les *props* ens permeten declarar components parametritzats:

Això es pot aprofitar per carregar components múltiples vegades, amb valors distints en cada cas.

```
function HomePage() {
        return (
            <div>
                <Header title="Bienvenido" user="Albert"/>
                <Header title="Hasta luego" user="Albert"/>
            </div>
        );
    }
```

10. Incloure codi JS a dins de fragments JSX.

És habitual que les *props* no es passin directament al contingut HTML, sinò que apareguin a dins de sentències més complexes.

Ejemplo:

```
function Header({title,user}) {
        return <h1>{title} {user ? user: 'usuario random'}</h1>;
    }
```

➔ **Quina és la notació per incloure JS a dins de fragments JSX?**

11. Iteració a dins de JSX:

Aquesta possibilitat d'incloure JS a dins de JXS permet que es puguin fer servir sentències i mètodes d'iteració.

Habitualment s'acostuma a fer amb el mètode *map*. Després veiem el problema del *for* en aquest context.

```
function HomePage() {
        const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton'];

        return (
            <div>
                {names.map((name)=>(
                    <Header title="Bienvenido" user={name}/>
                ))}
            </div>
        );
    }
```

Observa que l'aplicació carrega correctament i que es genera un component Header per a cada element de la llista *names*.

No obstant, la consola canta un avís:

react.development.js:199 Warning: Each child in a list should have a unique "key" prop.

Aquest avís es soluciona **afegint un *prop* anomenat *key*** que identifica unívocament cada component per a la llibreria React:

```
{names.map((name) => (
   <Header key={name} title="Bienvenido" user={name} />
))}
```

➔ **Què retorna el mètode *map*?**

11. El problema del *for* a dins de JSX:

JSX permet incloure sentències JS a dins de *{brackets}*, amb la condició que la sentència retorni un valor. Com hem vist, un *map* sí té un valor de retorn.

Però un *for* no té un valor de retorn.

Si volem fer servir *for* a dins dels *brackets*, hem de fer servir la tècnica de l'array auxiliar de sortida.

Aquesta seria la refactorització del codi anterior fent servir un for i un array auxiliar.

```
function HomePage() {
        const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton'];

        let components = [];
        for (let name of names) {
            components.push(<Header title="Bienvenido" user={name} />);
        }

        return (
            <div>
                {components}
            </div>
        );
    }
```

12. Gestió d'events:

JSX permet afegir handlers d'events (per a 'click', en l'exemple). Però s'han de fer servir atributs de l'element HTML.

Declara el nou component *LikeButton* i afegeix-lo a dins del component *HomePage*.

```
function LikeButton() {
        return <button onClick={handleClick}>Like</button>
    }

    function handleClick() {
        console.log("increment like count")
}


function HomePage() {
        const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton'];

        return (
            <div>
                {names.map((name) => (
                    <Header key={name} title="Bienvenido" user={name} />
                ))}
                <LikeButton />
            </div>
        );
```

➔ **Quina diferència observes entre l'atribut que fem servir per passar el *handler* de l'event a JSX i el que fem servir en HTML?**

➔ **Quin avantatge pot tenir declarar el *handler* a dins del propi component (en aquest cas, del component *LikeButton*).**

➔ **Com faries per passar-li al component *LikeButton* un *handler* que hem declarat fora del component?**

12. Gestió de l'estat d'un component:

Finalment, afegirem una **variable d'estat** al component *LikeButton*.

Per fer-ho, farem servir un *hook* de React: concretament el ***hook useState*** (ja veurem més endavant què és exactament un *hook* i quins altres *hooks* hi ha).

Modifica el codi de la següent manera:

```
function LikeButton() {
        const [likes, setLikes] = React.useState(0);

        return <button onClick={handleClick}>Like ({likes})</button>
    }
```

```
function LikeButton() {
        const [likes, setLikes] = React.useState(0);

        function handleClick() {
            setLikes(likes + 1);
        }

        return <button onClick={handleClick}>Like ({likes})</button>
    }
```

Observa que:

1. El *useState* és un mètode de l'objecte React.
2. La crida a *useState* retorna dos elements en un array:
    a. El primer és la **variable d'estat**. Emmagatzema el valor actual de l'estat.
    b. El segon és el **modificador de l'estat**. S'haurà de cridar sempre que es vulgui modificar el valor de la variable d'estat.

➔ **Si afegim a l'aplicació una altre instància del component *LikeButton*, el seu estat serà independent del del primer botó?**

➔ **Ara que el component *LikeButton* té estat, es veu més clarament per què el *handler* del clic s'ha declarat a dins del component?**

➔ **Pot el pare modificar directament el valor de l'estat d'un fill (mitjançant, en aquest cas, el setLIkes)?**

➔ **Se t'acut alguna manera de modificar des del pare l'estat del fill?**

➔ **Si el pare tingués un estat, com el passaria al seus fills?**