Resumen apuntes REACT

ReactJS

React es **una de las bibliotecas más populares de JavaScript para crear interfaces de usuario.** Es un proyecto *Open-Source* creado y mantenido por ***Facebook*** para el desarrollo de interfaces en el *Front-End*. Es decir, **es la capa de vista de una aplicación MVC (Modelo-Vista-Controlador).**

Características

- **Declarativo:** tu trabajo será especificar **qué contiene la interfaz** (no cómo se actualiza en cada momento).
- **Basado en componentes:** en React, todo es un componente, esto es, **una pieza de interfaz HTML reutilizable.** Puedes crear tus propias etiquetas HTML reutilizables.

- **Independencia de plataforma:** React puede ser utilizado para el **renderizado del lado del servidor con *Next***, para la **generación de sitios estáticos con *Gatsby*** o para el **desarrollo de aplicaciones móviles con *React Native*.** ¡Todo con el mismo lenguaje!

Virtual DOM

Si bien React nos abstrae del proceso de actualizar la interfaz a través de su lenguaje declarativo, conviene saber sobre qué estamos trabajando.

React opera bajo un patrón denominado ***Virtual DOM***. Un *Virtual DOM* es un objeto que **representa el estado deseado de la interfaz**, mientras que el *DOM* real es un objeto que representa el **estado actual** de la interfaz.

React observa cambios en determinadas variables (*`props` y `state`*) y, al haber un cambio en dichas variables, se busca la mejor forma de llegar al *DOM* deseado desde el *DOM* actual a través de un algoritmo denominado ***DOM Diffing***.

JSX

Otra característica de React es que posee **un lenguaje de desarrollo propio denominado *JSX*.** Este lenguaje es usado únicamente **en tiempo de desarrollo** ya que, al construir la aplicación, **todo código *JSX* es traducido en código *JS* procesable por el navegador mediante *Babel* y *Webpack*.**

Webpack

Webpack es un **empaquetador de módulos.** Significa que, con *Webpack*, puedes convertir **varios archivos separados en un solo archivo que haga exactamente lo mismo.** Por ejemplo:

Al trabajar con *Webpack*, podemos dividir el código en **varios archivos**. Solo podrás incluir (`import`) **código previamente exportado** (`export`) **por otro archivo**.

En este ejemplo, tienes **un archivo principal** (`index`) que requiere, es decir que importa, **dos variables de otro archivo.** Luego de eso, las usa para mostrar la suma por consola.

```
// variables.js

export const numero1 = 20;
export const numero2 = 30;


// index.js

import { numero1, numero2 } from './variables.js';

console.log("La suma es "+(numero1 + numero2));
```

Al llegar el momento de construir el programa, de generar el resultado final que será entregado, *Webpack* empieza a analizar **desde el punto de entrada** (normalmente es el archivo `index`). De ese archivo analiza los `imports` y de esos archivos requeridos sus `imports` y así arma el árbol de dependencias.

Una vez creado el árbol, genera un código que hace **exactamente** lo mismo, pero **ofuscado** y **optimizado** (lo empaqueta).

Babel

Si bien *Webpack* se encarga de compilar todos los archivos necesarios ¿Qué sucede con *JSX*? ¿Bastará con que se unan varios *JSX* en uno solo para que eso lo entienda el navegador?
La respuesta es que no. Para eso **necesitamos otro programa que trabaje junto con *Webpack* para convertir el código *JSX* en llamadas a `React.createElement`**. Esa herramienta es ***Babel*** y justamente esa es su función.

Vamos a usar *Babel* **en el momento en que desarrollemos en un código distinto al procesable por el navegador.**

Componentes

Cuando desarrollamos una interfaz, necesitamos gestionar **cuatro aspectos clave: el código *HTML*, la funcionalidad *JavaScript*, los estilos *CSS* y la información mostrada.**

De estos cuatro aspectos, salen **tres principios** del desarrollo en ReactJS.

1. **Pensar en componentes (HTML y JS):** un **componente** es **una pieza de código reutilizable que representa una porción de la interfaz**. Podemos pensar en los componentes como "*HTML en función de un objeto JavaScript*".
2. **Gestionar la apariencia (CSS)**: al fin y al cabo, estamos desarrollando front-end y es imprescindible **cuidar todo detalle de la experiencia del usuario**.
3. **Gestionar el estado (información)**: una interfaz React es una interfaz que cambia **cuando cambia la información contenida en ella sin recargar la página**.

Cómo dijimos antes, un componente es **un código *HTML* vinculado a un código *JS* que sirve para representar una porción de la interfaz.**

Que sea un código *HTML* vinculado a un código JS significa que el pedazo de código *HTML* de un componente cambiará **automáticamente y sin recargar la página cuando cambie su objeto *JavaScript* asociado**. Es decir, **reaccionará.**

De esta forma, puedes hacer cambios en la interfaz muy fácil: si cambias un dato, **de forma automática cambias la interfaz.** Eso se llama reactividad y los componentes nos sirven **para implementar esa característica.**

Para implementar esta reactividad es necesario introducir el concepto de **variable reactiva**. En React tenemos variables normales y también tenemos variables reactivas. Una variable reactiva es como una variable común con la salvedad de que **al cambiar su valor, cambian las porciones de interfaz que usen dicho valor, automáticamente.**

Las variables reactivas se gestionan únicamente a través de componentes. Existen **dos objetos *JavaScript*** a los que reacciona todo componente React:

- El objeto llamado **`Props`**: son datos reactivos **externos** al componente, es decir, **que sirven para la comunicación con otros componentes.**
- El objeto llamado **`State`**: son datos reactivos **internos** al componente, es decir, que solo sirven para reactividad y funcionalidades **dentro de este componente.**

La forma de gestionar las `props` y el `state` varía **según la estructura del componente.**

Podemos agrupar los componentes en:

Según su propósito

- **Contenedores:** un contenedor es **un componente que no influye en la interfaz**, sino que sirve **para dar características a otros componentes.**
- **De presentación:** un componente de presentación es de lo que venimos hablando, **una porción de tu interfaz *HTML* que va a reaccionar a los cambios de un objeto *JavaScript* específico.**

Según su estructura

- **Componente funcional:** se crea **con una función.** Las `props` **son su argumento** y el `state` **se gestiona mediante *Hooks*** (lo veremos más adelante).
- **Componente basado en clases:** se crea **con una clase.** Las `props` son **el argumento al constructor** y el `state` **se lee e inicializa con `this.state` y se cambia con `this.setState({})`**

Componente funcional

Un componente funcional es un componente que se crea **usando una función de *JavaScript*.** Veamos el paso a paso para crearlo:

1. Crea un **nuevo archivo** *.jsx* dentro de la carpeta **/src.**
2. Dentro de ese archivo, escribe **una función normal.**
3. El nombre de la función será el **nombre del componente.** Los componentes se deben escribir en ***CamelCase.***
4. Recibe **un solo parámetro** llamado **props**.
5. Escribe en la sentencia **`return` el código *JSX* que reaccionará a los cambios.** Recuerda siempre que debes retornar **un solo nodo raíz**.
6. Una vez creado tu componente, debes **exportarlo**. Coloca al final del archivo:

```
export default NombreDeTuComponente
```

7. Ya puedes incluir ese archivo y usar tu componente en *JSX* como si fuese **una nueva etiqueta *HTML*** (incluso pasarle atributos que serán recibidos a través de las *props*).

Ejemplo de componente funcional

Item.jsx U

src > Item.jsx > ...

```jsx
function Item(props)
{
    return (
        <div>
            <span>{ props.nombre }</span>
            <span>{ props.precio }</span>
        </div>
    );
}

export default Item;
```

Ejemplo de importación y uso del componente

Item.jsx U App.js M

src > App.js > ...

```jsx
import Item from './Item';
import './App.css';

function App() {
  return (
    <div className="App">
        <Item
            nombre="Celular"
            precio="2500"
        ></Item>
    </div>
  );
}

export default App;
```

Componente basado en clases

Un componente basado en clases **se crea mediante clases de *JavaScript ES6*.**

Veamos el paso a paso para crearlo:

1. Crea un **nuevo archivo .jsx** dentro de la carpeta **/src.**
2. Dentro de ese archivo, escribe ahora una **clase vacía.**
3. El nombre de la clase será el **nombre del componente.** Los componentes se deben escribir en ***CamelCase***. La clase debe heredar **de `React.Component`**. Necesitarás importar el paquete React.
4. Crea el método **`constructor`** y recibe allí **un solo parámetro llamado `props`**. Llama al constructor de la clase padre ahí mismo con **`super`**, pasado las props como parámetro.

5. Escribe el método **render**. El retorno de ese método será **el código *JSX* que reaccionará a los cambios.** Recuerda siempre que debes retornar **un solo nodo raíz.**

6. Una vez creado tu componente, tienes que exportarlo. Coloca al final del archivo:

```
export default NombreDeTuComponente
```

7. Ya puedes incluir ese archivo y usar tu componente en *JSX*, **como si fuese una nueva etiqueta *HTML*** (incluso pasarle atributos, que serán recibidos a través de las props).

Ejemplo de componente de Clase

```jsx
import React from "react";

class Item extends React.Component
{
    constructor(props)
    {
        super(props);
    }

    render()
    {
        return (
            <div>
                <span>{ this.props.nombre }</span>
                <span>{ this.props.precio }</span>
            </div>
        );
    }
}

export default Item;

```

Ejemplo importación y uso del componente

Item.jsx U | App.js M

src > App.js > ...

```jsx
import Item from './Item';
import './App.css';

function App() {
  return (
    <div className="App">
        <Item
            nombre="Celular"
            precio="2500"
        ></Item>
    </div>
  );
}

export default App;

```

NOTA IMPORTANTE:
En la materia Programación Visual – SEDE San Salvador de Jujuy y Escuela de Minas, se considera trabajar únicamente con los componentes funcionales.
Según plan de estudio vigente, el estudiante no ha visto el paradigma orientado a objetos ni el funcional. Como existe la materia programación orientada a objetos, en este espacio curricular se trabaja con una introducción a la programación funcional.

Pensando en ReactJS

Una de las grandes ventajas de React es cómo te hace **pensar acerca de la aplicación mientras la construyes**.

Este proceso se puede resumir en tres pasos:

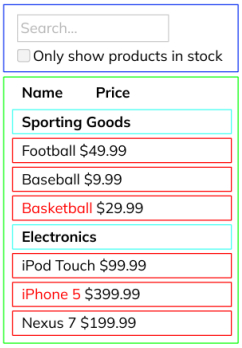
1. Pensar en Componentes.
2. Decide dónde debe vivir la información de la vista (estado).
3. Decide qué cambia cuando dicha información cambia.

Fuente bibliográfica: Pensar en React – React

1. Componentes

- Podemos empezar con un mock o un diseño estático de la vista que deseamos construir.
- Luego, será importante dividir esa interfaz en pequeñas cajitas y darle nombre. Esas cajitas serán tus componentes.
- ¿Cómo decidir qué es un componente? Puedes usar el Principio de Responsabilidad Única: *"Si tiene más de una función, se debe dividir."*
- Recuerda que los componentes muestran información, así que deben estar organizados según la información a mostrar.

Ejemplo Diseño componentes



- Una vez que hemos decidido los distintos componentes, el siguiente paso es crear una primera versión estática de dicha aplicación.
- Una versión estática es una versión de prueba en la que no hay interactividad ni conexión entre los componentes.
- Es importante separar estos dos procesos para hacer el desarrollo más mantenible.
- En resumen: luego de desarrollar un mock, desarrollamos una primera versión **sin interactividad.**

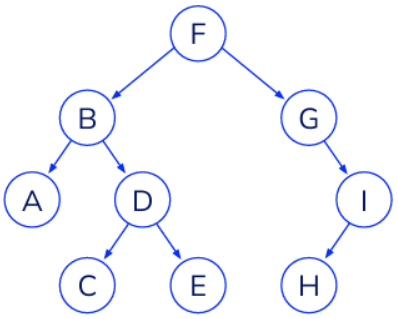
2. Decide dónde debe vivir el estado

- Para hacer tu interfaz de usuario interactiva necesitarás realizar cambios en tu modelo de datos interno. React lo logra gracias a su estado.
- Para descubrir la mínima versión del estado, sería importante recordar el DRY: *Don't Repeat Yourself.* Es decir, no repetir información.

- Para decidir si un dato es parte del estado puedes aplicar los siguientes criterios:
  - No debe venir como atributo del componente.
  - Debe cambiar con el tiempo.
  - No debe ser calculable en base a otro dato.

- Recuerda que, en *ReactJS*, la información fluye *en un solo sentido:* ***del componente padre a los componentes hijos.*** Tal como se muestra en el gráfico del ejemplo.
- Es importante decidir ***dónde*** debe vivir esa información.
- Identifica qué componentes muestran algo con base a este estado.
- Busca un componente común a estos más arriba en la jerarquía.
- Este componente o uno más arriba en la jerarquía debería poseer el estado.

- Si no puedes crear un nuevo componente que tenga sentido que posea el estado, crea un nuevo componente simplemente para poseer el estado y agrégalo en la jerarquía sobre los componentes que lo necesitan.



3. Decide qué cambia cuando cambia el estado

- Una aplicación interactiva en *ReactJS* cambia su comportamiento cuando el estado cambia.
- Decide qué componentes cambian cuando cambia el estado.
- Puedes hacerlo mediante escuchadores de eventos.
- Hay librerías como *Redux* que te permiten manejar el estado de forma centralizada.

Renderizando elementos

Mostrando componentes

Todo empieza en un solo lugar, no importa cuál sea el alcance de tu aplicación React, si estás haciendo un home banking, tu portfolio personal o una lista de compras. Toda aplicación React empieza en **un solo y único componente.**

En el archivo index.js (main.js con Vite), en el punto de entrada del programa, vemos la llamada a la función render de ReactDOM. Lo que hace esta función **es montar un componente en un elemento HTML** que es considerado **el componente raíz**. Como toda aplicación React se monta desde un único componente, la forma de mostrar elementos será añadiéndolos al árbol de componentes

Observa que hay un solo nodo raíz (en este caso es React.StrictMode).

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

Paso a paso para añadir un componente al árbol de componentes:

1. Define el **elemento** que quieres mostrar. Con React podemos renderizar componentes personalizados así como cualquier otra etiqueta *HTML*.
2. Define **dónde** quieres insertarlo, en qué parte del árbol: ¿Se trata de un componente que deba envolver a todos? ¿Es un ícono específico de un menú? ¿Dónde se mostrará esto?
3. Ve hacia el archivo **donde quieres insertar el elemento.**
4. Si quieres insertar un componente personalizado, asegúrate de primero **importarlo.**
5. Ve hacia la línea **donde quieres insertar el elemento.**
6. Escribe el nombre del componente **como si fuese una etiqueta *HTML*.**
7. Puedes pasarle **props como si fuesen atributos *HTML*.**
8. Un componente puede tener componentes hijos y serán pasados todos a través de la `prop children`.

Ejemplos de organizaciones de los componentes en el árbol de componentes:

```
function App() {
  return (
    <div className="fondo">
        <div>
            <h1>Mi aplicación</h1>
            <p>
                Hola mundo
            </p>
        </div>
    </div>
  );
}
```

```
function App() {
  return (
    <div className="fondo">
        <main>
            <h1>Mi aplicación</h1>
            <section>
                Hola mundo
            </section>
        </main>
    </div>
  );
}
```

Ejemplo de organización de componentes creados por separados dentro del árbol de componentes

```
import Contacto from "./components/contacto/contacto.component";
import Divider from "./components/divider/divider.component";
import "./fondo.css";

function App() {
  return (
    <div className="fondo">
        <Divider></Divider>
        <Contacto></Contacto>
        <Divider></Divider>
    </div>
  );
}
```

Una herramienta muy útil de *JSX* son **las llaves simples ({ y })**. Dentro de ellas podemos renderizar cualquier cosa (componentes, variables, etc.) **en una sola línea.** En otras palabras: podemos usar cualquier código *JS* / *JSX* dentro de estas llaves siempre y cuando nos ocupe **una sola sentencia de código** (no podemos hacer un `if-else`, pero sí un ternario, por ejemplo).

```
function App(props) {
  return (
    <div className="fondo">
        { props.isAuth ? "logueado" : "no logueado" }
    </div>
  );
}
```

Propiedades (props)

Las **propiedades** (o simplemente "props") son **un objeto que se pasa como argumento al momento de renderizar un nuevo componente React.** Este objeto permite al componente **recibir información externa e interactuar con otros componentes.**

Las *props* tienen una importancia central en todo el desarrollo React ya que permiten **el desacoplamiento de características y la reutilización de código.**

¿Qué podemos encontrar dentro de este objeto?

- **Mapeo de atributos *HTML*:** todo componente React (sea personalizado o basado en etiquetas *HTML* estándar) **soporta también los atributos *HTML* como `style` o `id`.** Algunos atributos tienen un nombre ligeramente diferente (en vez de `class`, usamos `className`).
- **Propiedades específicas de React:** React puede insertar propiedades dentro del objeto *props* **de forma automática.** Un ejemplo es `children`, que contiene todos los elementos que se pasan como hijos al componente actual.

- **Propiedades personalizadas:** podemos pasar nuestras propias props al escribirlas como **atributos *HTML* del componente.**
- **Información inyectada de forma externa al componente:** muchas veces necesitamos dotar a un componente de **características específicas.** Eso lo lograremos a través de las props.

Ejemplo

Establecer la propiedad className a una etiqueta div (que suplanta al class) para aplicar un estilo

```
<div className="miclase">
Contenido
</div>
```

Estado de un componente

Se define como **estado al conjunto de variables reactivas encapsuladas dentro del componente.** En otras palabras: el estado son **variables que controlan el cambio de la interfaz** pero solo **en el componente en el que se encuentran** (y sus componentes hijos).

El estado permite implementar reactividad **focalizada** ¿Qué significa esto? Esto significa que me sirve para **implementar cambios en la interfaz que afecten solo a un elemento concreto o a sus hijos.**

Si quiero mostrar un submenú al hacer clic en un ítem, eso debería controlarse mediante el estado porque es un cambio focalizado. Si, en cambio, quiero cambiar todo el tema del sitio a modo oscuro, eso debería gestionarlo mediante props ya que es un cambio compartido por muchos componentes.

Comparación entre estado y propiedades

Estado	Propiedades
Gestiona cambios **puntuales** a un componente o sus hijos.	Gestiona cambios **globales** a varias partes de la interfaz.
Se crea **dentro del componente**.	Se pasa **por fuera del componente**.
No se comparte directamente con otros elementos.	Pueden contener **datos compartidos con otros elementos**.

- Los componentes funcionales **no tienen acceso al estado directamente**.
- Para acceder al estado desde un componente funcional se usan ***hooks***.
- Los *hooks* son una forma de **desacoplar características en React**. A diferencia de las clases, que tienen en **`React.Component`** todo lo necesario, los componentes funcionales **deben requerir las cosas que van necesitando mediante *hooks***.

- Para usar el estado vamos a usar el *hook* **useState**, importado desde **React**.
- `UseState` es una función que **recibe un parámetro** (el valor inicial de esa variable reactiva) y **devuelve un `array`**.
- El primer ítem de ese `array` es **una variable** y el segundo es **una función**. Cuando se llama a la función y se le pasa un valor, ésta **cambia el valor de la variable** y a su vez **cambia la porción de la interfaz en la que se use dicha variable**.

Eventos

Un evento es un suceso importante en la interfaz. Con esto nos referimos a que **debe ser atendido y la interfaz debe ofrecer una respuesta.**

La gestión de eventos tiene varias partes:

1. Por un lado, tenemos los **eventos** propiamente dichos, acontecimientos que ocurren **en nuestra interfaz.**
2. Por otro lado, tenemos los **escuchadores,** que son **porciones de código que atienden a un evento y ejecutan una respuesta personalizada al ocurrir dicho evento.**
4. Finalmente, los **disparadores,** son **las condiciones para que un evento ocurra.**

React nos ofrece la posibilidad de utilizar los mismos eventos que usamos en *HTML* y, además, **crear nuestros propios eventos.**

En React, toda la gestión de eventos se hace a través de *JSX* mediante **atributos específicos.** Estos atributos reciben como valor **funciones** que serán ejecutadas **cuando el evento se dispare.**

Veamos algunos de los eventos más utilizados en el desarrollo con React.

Ciclo de vida

El ciclo de vida de un componente es, como bien indica su nombre, el conjunto de pasos secuenciales que transcurren **desde que se inserta un componente en la interfaz hasta que es quitado y limpiada la memoria.**

A grandes rasgos, cuando se trata de gestionar a los componentes de una interfaz, React tiene tres tareas principales:

1. La primera tarea es el **montado.** Montar un componente significa **insertarlo en el *VirtualDOM* de la página.**
2. Una vez que el componente está montado, debe reaccionar a los cambios en los datos a través de la **actualización.** La actualización se dispara **al cambiar el estado o el valor de las *props* de un componente** y ocasiona que **se vuelva a procesar con los nuevos datos.**
3. Finalmente, cuando el componente debe ser quitado de la interfaz, se procede con el **desmontado.** En esta fase el componente se quita y se liberan los recursos utilizados.

Ciclo de vida en componentes funcionales

Montado

- El montado de un componente funcional es más sencillo. Simplemente **se llama a la función y el retorno de la función es el componente que se inserta en el *DOM*.**
- Es importante recordar que React lee la función **una sola vez al inicio.** El montado, en este caso, será **el único lugar donde se procese un código diferente al que está retornado** (el cuerpo de la función).
- También es el lugar para **inicializar todos los hooks necesarios.**

Actualización

- La actualización es ocasionada por **un cambio de estado o de props.**
- Consiste en volver a ejecutar **el elemento *JSX* retornado por la función** (y no toda la misma).

Desmontado

El desmontado es, también, más sencillo. Consiste simplemente en **la liberación de forma automática de todos los recursos consumidos por la función.**

Algunos consejos

- La gestión del ciclo de vida con componentes funcionales es **mucho más simple y fácil.**
- No siempre simple significa mejor. El costo de esa simplicidad se cobra haciendo que la personalización de determinadas partes del ciclo de vida sea **mucho más costosa,** por no decir imposible.
- El ciclo de vida de un componente funcional es **una reducción del ciclo de vida de un componente basado en clases.**
- Para ocasionar una actualización, usa el estado mediante **`useState`.**

- Para añadir comportamientos asincrónicos usa el hook **`useEffect`** que sirve justamente para ese fin. Este hook permite añadir comportamientos asincrónicos **protegiendo la creación y el montado.** Cuando se termine de montar en el *DOM*, se llaman todos los `callbacks` registrados con `useEffect`, por lo que tiene un comportamiento parecido a **`componentDidMount`.**

```jsx
import React, { useEffect, useState } from "react";

function App(props) {
    const [user, setUser] = useState({});

    useEffect(function () {
        fetch(`http://api.miistio.com/users/${this.props.id}`)
            .then((r) => r.json())
            .then((user) => setUser(user));
    });

    return (
        <div>
            <span>{user.nombre}</span>
            <span>{user.apellido}</span>
        </div>
    );
}

export default App;
```

React Router

React Router es una completa **librería para implementar enrutamiento del lado del cliente y del lado del servidor para aplicaciones React**.

Esta librería nos servirá para **implementar enrutamiento** o, en otras palabras, **renderizar ciertos componentes en función de eventos en la *URL*** (como el cambio en la url o en alguno de sus parámetros).

Cuándo puedes usarla

- ¿Quieres **renderizar un componente si hay una *URL* específica**?
  Entonces te sirve *React-Router.*
- ¿Quieres **actualizar un componente en función de una *query* de la *URL***?
  Entonces te sirve *React-Router.*
- ¿Quieres **hacer zonas de acceso restringido en tu aplicación**?
  Entonces te sirve *React-Router.*

SPA Routing

React-Router es una capa de conexión (un adaptador) de las funcionalidades de enrutamiento para React.

El enrutamiento del lado del cliente (o *Single Page Application Routing*) consiste en **mostrar determinada interfaz en función de la *URL* sin que haya solicitudes a un servidor,** sino de forma **dinámica** mediante JavaScript.

Para lograr esto, JavaScript ofrece las siguientes posibilidades de forma nativa:

- Usar **el hash (#) de la *URL*** y, en base a eso, **definir la interfaz a visualizar.**
- Usar la ***API History*** que permite **gestionar las *URL* sin necesidad de usar el hash,** de una forma más limpia.

Paso a paso

1. Crea un **nuevo proyecto React o escoge uno ya existente.**
2. Abre la línea de comandos en ese proyecto para instalar **React-Router por npm.**
3. Ejecuta **el siguiente comando:**

```
npm install react-router-dom
```

4. Existen varios adaptadores en React-Router. **El adaptador al *DOM* sirve para aplicaciones SPA.**

Componentes de React Router

Vamos a agrupar los componentes ofrecidos por *React Router* en las siguientes categorías:

- **Routers:** sirven para determinar **el medio que vamos a usar para el enrutamiento** (`hash` / `history` / etc.).
- **Diccionarios de rutas:** componentes que sirven para **definir el diccionario de rutas de la aplicación.**
- **Navegación:** componentes que sirven para **navegar a una ruta específica.**

BrowserRouter

Este router es el medio que se recomienda para la navegación en ***SPA* con React Router.**

El `BrowserRouter` **almacena la información de la ruta actual en memoria, proporcionando una *URL* limpia.** Sirve para la mayoría de los casos. Es el router que usamos **por defecto.**

Sin embargo, hay que tener en cuenta que, al no usar `hash` ni cualquier otro añadido a la *URL*, es necesario contar con un servidor que contemple el **`fallback`** (lo veremos más adelante en esta clase). Resuelto eso, el `BrowserRouter` es la mejor alternativa.

Hooks en React Router

Los *hooks* de React Router permiten ir añadiendo **características puntuales a un componente funcional.**

Como es de esperarse, estas características están relacionadas **con la comunicación con la ruta actual.** Vamos a ver los *hooks* más comunes en el desarrollo con React-Router.

UseRoutes

El *hook* **`useRoutes`** es un reemplazo al **`Routes`** y **`Route`**. Sirve para implementar la misma lógica que esos dos componentes pero de forma **programática** en JavaScript (en vez de la forma declarativa con *JSX*).

Este *hook* recibe como argumento un `array` de objetos. Cada objeto puede tener **las mismas propiedades que un `Route`,** incluyendo la propiedad **`Children`**. Piensa en este objeto como las *props* que se le pasarían a un `Route`.
El hook `useRoutes` retorna **un elemento de React listo para ser renderizado en *JSX* o retornado directamente por un componente.**

```jsx
import { useRoutes } from "react-router-dom";

function App() {
  let routes = useRoutes([
    {
      path: "/",
      element: <Dashboard />,
      children: [
        {
          path: "messages",
          element: <DashboardMessages />
        },
        { path: "tasks", element: <DashboardTasks /> }
      ]
    },
    { path: "team", element: <AboutPage /> }
  ]);

  return routes;
}

export default App;
```

UseNavigate

El *hook* **`useNavigate`** retorna **una función que hace exactamente lo mismo que el componente `Navigate`.**

Nos puede ser muy útil cuando necesitamos realizar **una redirección de forma programática en lugar de *JSX*** (como por ejemplo en el *handler* de un evento).

```jsx
import { useNavigate } from "react-router-dom";

function SignupForm() {
  let navigate = useNavigate();

  async function handleSubmit(event) {
    event.preventDefault();
    await submitForm(event.target);
    navigate("../success", { replace: true });
  }

  return <form onSubmit={handleSubmit}>{/* ... */}</form>;
}

export default SignupForm;
```

UseParams

El *hook* **`useParams`** permite **acceder a los parámetros pasados por la *URL*** ¿Recuerdas que, cuando vimos el `Route`, comentamos que se pueden pasar parámetros anteponiendo dos puntos (`:`) en un segmento de la *URL*? Este *hook* es la otra cara de la moneda. Nos permite obtener dichos parámetros **en un componente.**

El *hook* retorna un objeto JavaScript plano donde las claves son **los nombres definidos en la ruta** (los segmentos que empiecen con dos puntos) y **los valores son los caracteres ubicados en esa posición de la ruta.**

```jsx
import { useEffect, useState } from 'react';
import { Routes, Route, useParams } from 'react-router-dom';

function ProfilePage() {
  let { userId } = useParams();
  const [user, setUser] = useState({});

  useEffect(() => {
    fetch(`/users/${userId}`)
        .then(r => r.json())
        .then(d => setUser(d))
  });

  return (
      <div>{user.name}</div>
  );
}

function App() {
  return (
    <Routes>
      <Route path="users">
        <Route path=":userId" element={<ProfilePage />} />
      </Route>
    </Routes>
  );
}

export default App;
```