Universidad Tecnológica Nacional

Facultad Regional Rosario

Tecnicatura Universitaria de la Programación

Programación III

Unidad 1.1

Lógica de componentes

Conceptos básicos del código de React

Lógica de componentes

Recordemos de la primera clase, como es qué React se nos presenta:

*La biblioteca para **interfaces de usuario** web y nativas*

Esto está muy bien pero, ¿No es acaso también posible construir interfaces de usuario con **HTML**, **CSS** y **JS**? Efectivamente, pero lo que nos permiten las librerías como **React** es construir interfaces de usuario **complejas, interactivas y reactivas al usuario**, de una manera mucho más simple que sólo con las 3 herramientas nombradas anteriormente.

¿Cuál es el secreto de React para simplificar estos apartados complejos? React utiliza lo denominado **lógica de componentes**. Todas las interfaces de usuario, al final del día, están compuestas por componentes.

¿Qué son los componentes? Son bloques de código que funcionan como unidad principal de trabajo en la librería React. Se trabaja con componentes por razones principales:

- **Reusabilidad**: después de crear un componente el mismo puede ser usado en distintas partes del sistema web. Esto nos ayuda a evitar repeticiones.
- **Separación de funcionalidades**: ya que cada componente podrá tener su lógica interna, evitaremos problemas comunes como demasiado código en un solo archivo.

¿Cómo se construyen los componentes? Como toda interfaz de usuario moderna, los componentes están compuestos por **HTML5**, **CSS3** y **JS.** La diferencia es que React utiliza un **acercamiento declarativo** en la construcción de estos componentes, donde nosotros definiremos el estado deseado de los elementos en el sitio web, y React se encargará de traducirlos al mismo. Es como si nosotros construyéramos nuestros propios elementos HTML.

Creando un nuevo proyecto de React

Para la creación de nuevo proyecto React utilizaremos la herramienta vite, con la cuál con una sóla línea de comando podremos tener listo un nuevo proyecto, incluyendo las optimizaciones necesarias, la posibilidad incorporada de salida a producción y un servidor de desarrollo local donde podremos ver los cambios que realicemos a nuestro sitio web en tiempo real.

Para poder correr la anterior línea de comando, debemos instalar (o tener instalado) Node.js en la computadora. Esto no es necesario para correr el código (Node.js nos sirve para correr código JavaScript fuera del navegador, pero React funciona **en el navegador**) sino para poder realizar las instalaciones de los paquetes (como *vite*) mediante **npm (node package manager).**

Vamos a abrir una terminal en la computadora y vamos a dirigirnos a la carpeta donde queremos instalar el proyecto. Cuando estemos allí, correremos el siguiente comando en consola:

npm create vite@latest

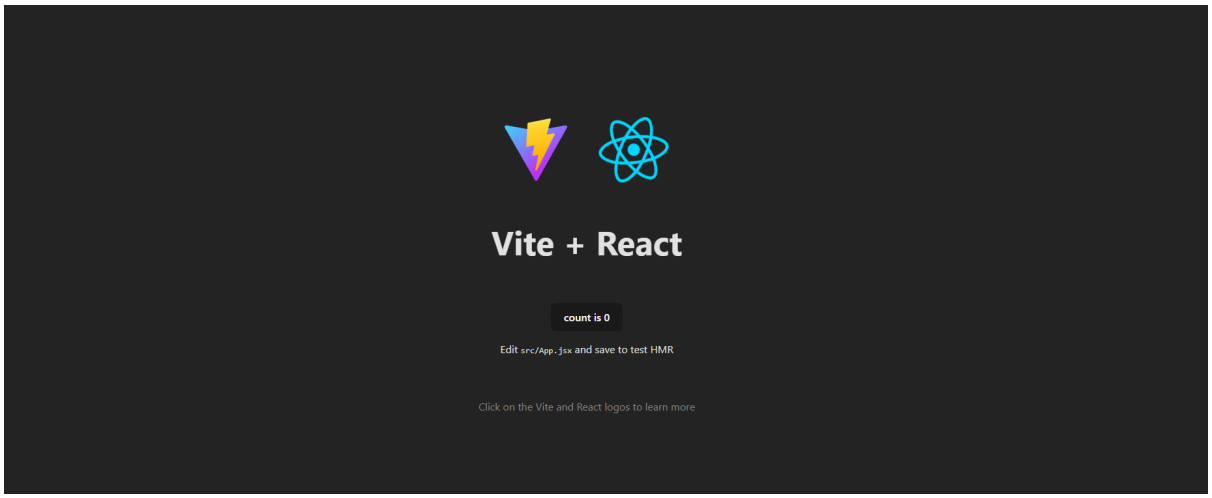
Nos va pedir el nombre que deseemos para nuestro proyecto.

Luego nos ofrece una serie de opciones, elegiremos React y Javascript.

Luego de terminada la instalación, correremos los siguientes comandos:

cd books-champion
npm install
npm run dev

Se nos abrirá una página *custom* de bienvenida de React, en caso de que todo haya sido instalado correctamente.



Ahora, vamos a editar el código de esta página para comprobar su funcionamiento. Abrimos la carpeta en VSC (podemos en la misma carpeta abrir una terminal y tipear *code .* (punto)).

Allí veremos varios archivos:

- ***package.json***, contiene la información de los paquetes instalados (nombre y versionados), junto con las dependencias instaladas.
- La carpeta **src** (*source*), donde se encuentra el código fuente de toda la página.
- La carpeta **node_modules**, donde están instalados todos los paquetes para que funcione React (y donde se van a instalar las futuras librerías que usemos). El nombre de la misma aparece en gris porque esta carpeta **no debemos** subirla a git.

Realizaremos un poco de limpieza en la carpeta *src* para comenzar nuestro proyecto:

1. En la carpeta *src*, solo dejaremos los archivos: *App.jsx*, *index.css* e *main.jsx.*
2. Dentro del archivo *App.jsx*, borraremos todo lo que se encuentra dentro del *return* **menos** el div principal (borrar su atributo de clase igualmente). Borraremos también ambos *import.*
3. Agregaremos dentro de dicho div un h2 que diga "Books Champion App".
4. En el archivo *index.css* copiaremos el siguiente código: Código CSS

```css
@import url("https://fonts.googleapis.com/css2?family=Noto+Sans+JP:wght@400;700&display=swap");

* {
  box-sizing: border-box;
}

html {
  font-family: "Noto Sans JP", sans-serif;
}

body {
  margin: 0;
}

```


Analizando un proyecto de React

Si bien casi siempre hablamos de "este sistema web está escrito en React", "esta aplicación fue construida con React" es importante notar que los componentes **siempre van a estar escrito en JavaScript** (o TypeScript, si lo preferimos). De React vamos a aprovecharnos de parte de su sintaxis propia y de sus funcionalidades (métodos del ciclo de vida, *hooks*, etc)

El archivo *index.js* será el archivo que será servido al servidor y será el que veamos en *localhost:5173,* sin embargo, el archivo no será servido como está escrito allí, sino que React se encargará de optimizarlo y traducirlo no sólo para que corra en el navegador instalado (como por ejemplo Google Chrome) sino en todo tipo de navegadores junto también a versiones anteriores de lo mismo.

Buscamos siempre entonces **que nuestro sistema web sea compatible con la mayor cantidad de navegadores y versiones de los mismos.**

Este proceso de traducción y optimización es disparado por el comando *npm run dev*, realizado antes de servir nuestra aplicación en el servidor de desarrollo local. Esto se puede ver en por ejemplo la línea de código:

import 'index.css';

Ya que es imposible importar un archivo entero de css a otro de JS. O por ejemplo en la sintaxis de <*App* />, que tampoco es sintaxis nativa de JavaScript.

Vamos a explicar un poco el siguiente código de *index.js*:

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

Los primeros dos *import* importan al archivo los objetos que representarán a las librerías React y ReactDOM. Ambas en conjunto forman React, solo que están separadas por cuestiones de funcionalidad. Podemos visualizar que ambas se encuentran listadas en *package.json*.

ReactDOM va a crear la *root* donde colocará toda la aplicación (en este caso es un elemento del documento *index.html* que tiene la id *root*). Luego en *root.render()* renderiza la aplicación entera.

Allí, vemos por qué importamos React, en donde dice *React.StrictMode*, este componente nos permitirá reforzar código limpio y que siga las buenas prácticas de React, dándonos advertencias en consola cuando no logramos hacerlo. Se aplicará sobre todos los componentes que encierre (en este caso, App).

En la cuarta línea, vemos que importamos el componente "App" de './App', el ./ significa que nos paramos en la misma carpeta, y buscamos el archivo *App.js* (no es necesario agregar el js en React). Luego, a ese componente lo vemos insertado en el primer atributo como si fuera un elemento HTML (<App />)

Ahora, exploremos el archivo *App.js*:

```
1  import './App.css'
2
3  const App = () => {
4    return <h2>Book champions app</h2>
5  }
6
7  export default App
8
```

Vemos que es una función común de JS, llamada App (notemos que no usamos *camelCase* para el nombre de los componentes) que retorna una sintaxis extraña y claramente no válida en JS. Además de eso, vemos que se exporta el componente por *default* (y recordemos de la sección de JS moderno, por eso lo importamos como lo importamos en *index.js*).

Pero, ¿Qué significa esa sintaxis tan extraña y cómo podemos interpretarla?

JSX

JSX es un acrónimo de JavaScript XML, ya que al fin de cuentas, HTML y XML son descendientes del mismo lenguaje (SGML). Entonces, gracias a React podemos tener código HTML en nuestros archivos JS, ya que esto no funcionará si React no realizará toda la traducción y optimización del código.

Nota: Intentaremos llamar JSX siempre que podamos, aunque es común entre los desarrolladores react decirle HTML al JSX retornado por las funciones

¿Cómo funciona React entonces? El código JSX y la lógica del componente es el estado **deseado** que queremos ver en el navegador, y React se encarga de presentarlo al servidor.

Si abajo del h2 agregamos la siguiente línea de código:

* ¡Quiero leer libros! *

Al guardar, chequeamos la página y vemos que automáticamente se agregó el párrafo deseado.

Comparemos lo que nos costaría realizar el mismo proceso en JavaScript normal:

const parag = document.createElement('p');

parag.textContext = '¡Quiero leer libros!';
document.getElementById('root').append(parag);

Como podemos imaginar, a medida que construyamos sistemas web más complejos, con mayor interacción con el usuario y estados cambiantes (cambio de los estilos, aparición y desaparición de elementos, etc), escribir código de esta manera se volvería tedioso y difícil de llevar seguimiento.

Construyendo nuestro primer componente

Creación

Es práctica común poner los nuevos componentes en su propia carpeta dentro de *src*. Esto se hace para evitar meter todos los componentes al mismo nivel, y de esta forma ser más organizados. El componente "App" si se deja en el lugar que fue creado, ya que es un componente especial al ser el componente **raíz**, es decir, todos los componentes van a ser hijos de este componente, en forma tipo árbol.

Creamos entonces la carpeta *components*, con la carpeta *bookItem* (cada componente debe encontrarse dentro de su respectiva carpeta en *camelcase*) y dentro agregaremos un archivo llamado *BookItem.jsx*. Los componentes, en líneas generales, se suelen llamar con mayúscula inicial en cada palabra.

Dentro de este archivo escribiremos nuestro primer componente, pero, ¿Cómo? Debemos recordar que todos los componentes son funciones (al menos, a partir de React 16) que retornan el jsx que necesitamos. Probemos entonces con:

```
1  const BookItem = () => {
2      return (
3          <div>BookItem</div>
4      )
5  }
6
7  export default BookItem
```

Ese componente entonces retorna el texto "BookItem" dentro de un div. Luego, debemos exportar el componente para que lo podamos usar en otros archivos.

Iremos al componente "App" e importamos "BookItem", de la siguiente manera:

```jsx
import BookItem from './components/bookItem/BookItem'

import './App.css'

const App = () => {
  return (
    <div>
      <h2>Book champions app</h2>
      <p>¡Quiero leer libros!</p>
      <BookItem />
    </div>
  )
}

export default App
```

A *BookItem* lo escribimos simplemente como si fuera una etiqueta cerrada de HTML. Aquí vemos porque son importantes las mayúsculas al momento de nombrar el componente. Sin ellas, React consideraría que es un elemento nativo de HTML (como h2, p, div) y no sabría como interpretarlo, ni hacer la relación con nuestro componente *custom*.

Refrescamos la página y vemos que se ha agregado el componente a la pantalla del usuario.

¡Hemos creado nuestro primer componente! Vamos a agregar más información a "BookItem".

Books Champion App

¡Quiero leer libros!

BookItem

Al jsx que retorna, agreguemosle un h2 para el título, un h3 para el autor, un div que contenga la puntuación y un elemento p para la cantidad de páginas.

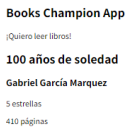
```
 @ ./src/App.js 4:0-45 21:35-43
 @ ./src/index.js 7:0-24 10:33-36

ERROR in src\components\BookItem.js
  Line 3:2:  Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (3:2)

webpack 5.70.0 compiled with 2 errors in 503 ms
```

Vemos que el IDE nos da error. Esto es debido a una de las reglas más importantes de React: todo jsx que es retornado debe tener un solo elemento raíz. En nuestro caso, al tener dos *divs* al mismo nivel como raíz, nos da un error. Más adelante, pensaremos una forma más prolija y moderna de solucionar este error, pero por ahora con encerrar todo nuestro código en un *div*, nos basta.

Podríamos cambiar los contenidos de las etiquetas por algo que refleje un verdadero "BookItem", quedandonos así:

```jsx
const BookItem = () => {
    return (
        <div>
            <h2>100 años de soledad</h2>
            <h3>Gabriel García Marquez</h3>
            <div>5 estrellas</div>
            <p>410 páginas</p>
        </div>
    )
}

export default BookItem
```

Vemos que a niveles más adentro del árbol, no hay problemas en poner dos elementos al mismo nivel, es solo la etiqueta raíz la que debe estar sola.



Mostrando valores dinámicos en pantalla

Como se estará imaginando el estudiante, no sólo tendremos un libro leído en nuestra aplicación y, sobre todo, los datos de ese libro no estarán *hardcoded* en el mismo HTML, sino que sus valores serán dinámicos. Deseamos entonces poseer múltiples componentes con datos que pueden cambiar en el tiempo. Para ello, vamos a crear variables que guardarán los valores actuales de nuestro libro, de la siguiente manera:

```
const bookTitle = "100 años de soledad";
const bookAuthor = "Gabriel García Marquez";
const bookRating = 5;
const bookPages = 410;
```

¿Cómo haremos para que React muestre el contenido de nuestras variables en la pantalla? Sencillamente, utilizaremos llaves *{ }* , las cuales nos permiten poner código JS que se resuelve **en un valor.** Por ejemplo, si dentro del *h2* colocamos { 1 + 1}, en pantalla nos aparecerá el número 2. Si ponemos { 5>6 ? 'Yes' : 'No' } esa expresión ternaria se resuelve por No, y es lo que veremos en pantalla.

Escribiremos el código entonces de la siguiente forma:

```jsx
const BookItem = () => {
    const bookTitle = "100 años de soledad";
    const bookAuthor = "Gabriel García Marquez";
    const bookRating = 5;
    const bookPages = 410;
    return (
        <div>
            <h2>{bookTitle}</h2>
            <h3>{bookAuthor}</h3>
            <div>{bookRating} estrellas</div>
            <p>{bookPages} páginas</p>
        </div>
    )
}

export default BookItem;
```

Nota 2: podemos combinar (interpolar) salida común con expresiones JS, como es el caso de la cantidad de páginas.

props en React

Pasando data mediante props

Ya podemos observar la re-usabilidad de los componentes copiando y pegando *<BookItem/>* varias veces, y veremos allí en la pantalla que el componente se multiplica. Si bien esto nos permite utilizar el componente varias veces y en distintas partes, no es realmente "reusable" ya que la data que estamos imprimiendo en pantalla es siempre la misma.

¿Cómo podemos aprovechar al máximo esta funcionalidad entonces? Utilizaremos el concepto similar al de parámetros dentro de las funciones de JavaScript, llamado *props*, (*properties*) que nos permitirá pasar diferente data a los componentes.

Veamos un ejemplo, si tuviéramos el componente "App":

```
const App = ()=>{
 const goalItem = '¡Recibirme!'
return <CourseGoals/>
}
```

Y quisiéramos pasárselo al componente "CourseGoals":

```
const CourseGoals = () =>
 {
   return (<ul>
                    <li>{goalItem}</li>
          </ul>) }
```

No podríamos hacerlo directamente, ya que "CourseGoals" no conoce a la variable *goalItem*. Lo que sí podemos hacer es *pasarsela* al componente hijo desde el componente padre, como una propiedad, de la siguiente manera:

```
const App = ()=>{
 const goalItem = '¡Recibirme!'
return
<CourseGoals goal={goalItem}/>
 }
```

Como si fuera un atributo de la etiqueta HTML. Luego, en el componente hijo:

```
const CourseGoals = (props) =>
{ return
(<ul>
```

```
        <li>{props.goal}</li>
</ul>)
}
```

props es el objeto que viene incluido en todo componente de React, puede poseer el nombre que queramos pero en general se lo llama *props*. De esta manera, el componente hijo puede acceder a atributos que posee el componente padre.

Veámoslo en nuestro proyecto:

Vamos a copiar y pegar el componente "BookItem" cuatro veces. Luego vamos a crear una variable en "App" llamada simplemente *books*, que será un **arreglo de objetos** con cuatro objetos: nuestro libro que creamos en el componente "BookItem" y tres más que se nos ocurran. Todos los objetos poseerán seis atributos: *title*, *author, rating, pageCount,imageUrl,available*.

Link con el archivo txt de Books

Ahora, pasaremos estos valores a los cuatro componentes, de esta manera tendremos *props* dinámicas generadas en JS y no *hardcoded*.

```
      <p>¡Quiero leer libros!</p>
      <BookItem
        title={books[0].title}
        author={books[0].author}
        rating={books[0].rating}
        pageCount={books[0].pageCount}
        imageUrl={books[0].imageUrl}
        available={books[0].available}
      />
      <BookItem
        title={books[1].title}
        author={books[1].author}
        rating={books[1].rating}
        pageCount={books[1].pageCount}
        imageUrl={books[1].imageUrl}
        available={books[1].available}
      />
      <BookItem
        title={books[2].title}
        author={books[2].author}
        rating={books[2].rating}
        pageCount={books[2].pageCount}
        imageUrl={books[2].imageUrl}
        available={books[2].available}
      />
      <BookItem
        title={books[3].title}
        author={books[3].author}
        rating={books[3].rating}
        pageCount={books[3].pageCount}
        imageUrl={books[3].imageUrl}
        available={books[3].available}
      />
    </div>
  )
```

Ahora, veremos en pantalla si se aplicaron los diferentes títulos, cantidad de páginas y puntaje. Las imágenes las implementaremos en la sección de la librería de estilos,

Vemos que no se aplicaron. ¿Por qué es eso? Definitivamente es porque no los declaramos en el componente "BookItem" mediante el objeto *props*. Además de aplicarlo, borraremos la lógica antigua del componente y haremos *destructuring* sobre *props* para obtener de manera más limpia las propiedades.

```jsx
const BookItem = ({ title, author, rating, pageCount, imageUrl, available }) => {
    return (
        <div>
            <h2>{title}</h2>
            <h3>{author}</h3>
            <div>{rating} estrellas</div>
            <p>{pageCount} páginas</p>
            <p>{available ? "Disponible" : "Reservado"}</p>
        </div>
    )
}

export default BookItem;

```

Y allí si observaremos que poseemos datos distintos en cada uno de los componentes.

imageUrl nos la remarca en rojo ya que no la estamos utilizando. Si nos marca todas las propiedades en rojo (pidiendo que usemos *prop-types*) la cátedra recomienda ignorar esta regla de *eslint* modificando el archivo *eslint.config.js* con la siguiente linea:

'react/prop-types': 'off',

Bootstrap 5

Agregando *bootstrap 5* al proyecto

En esta cátedra, utilizaremos *bootstrap* 5 como librería de estilizado en nuestro proyecto. Esto nos traerá beneficios y facilidades a la hora de estilizar, ya que en vez de escribir nuestras propias clases de CSS, podremos utilizar las *custom* que nos ofrece bootstrap.

Primero, instalaremos la librería react-bootstrap, que nos ayuda a integrar bootstrap con react (principalmente el uso de componentes personalizados de React).

En la terminal, pondremos el siguiente comando:

npm i react-bootstrap bootstrap

Verificamos en *package.json* que hayan sido instalados ambos paquetes. Luego, en *index.html* agregamos el link a el estilizado de bootstrap:

```
<link
 rel="stylesheet"

href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.
css"

integrity="sha384-9ndCyUaIbzAi2FUVXJi0CjmCapSmO7SnpJef0486qhLnuZ2cdeRhO02i
uK6FUUVM"
 crossorigin="anonymous"
/>
```

Esto nos permite utilizar las clases de bootstrap para estilizar, por ejemplo:

https://getbootstrap.com/docs/5.0/utilities/spacing/
https://getbootstrap.com/docs/5.0/utilities/flex/
https://getbootstrap.com/docs/5.0/utilities/colors/

Implementación de bootstrap 5

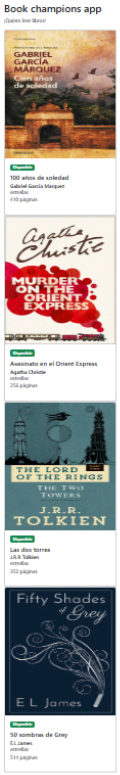
Vamos a crear tarjetas para nuestros libros, con una implementación similar a la que vemos en el primer ejemplo de Cards:

```jsx
import { Badge, Card, Button } from "react-bootstrap";

const BookItem = ({ title, author, rating, pageCount, imageUrl, available }) => {

    return (
        <Card style={{ width: "22rem" }} className="mx-3">
            <Card.Img
                height={400}
                variant="top"
                src={imageUrl !== "" ? imageUrl : "https://bit.ly/47NylZk"}
            />
            <Card.Body>
                <div className="mb-2">
                    {available ?
                        <Badge bg="success">Disponible</Badge>
                        :
                        <Badge bg="danger">Reservado</Badge>
                    }
                </div>
                <Card.Title>{title}</Card.Title>
                <Card.Subtitle>{author}</Card.Subtitle>
                <div>{rating} estrella{rating > 1 ? 's' : ''}</div>
                <p>{pageCount} páginas</p>
                <Button >
                    Actualizar titulo
                </Button>
            </Card.Body>
        </Card>

    )
}
```

Analicemos un poco el código que obtenemos ahí. Tenemos un componente de *react-bootstrap* llamado Card, que a su vez tiene subllamados a componentes internos Body, Title, Img). Fuimos repartiendo según el enmaquetado necesario las diferentes *props* de nuestro componente, además de agregarle una imágen como cabezal (utilizamos el operador ternario que indica **que en el caso que la variable imageUrl no posea valor, usamos una imagen *default***) Agregamos además un componente Badge, que en el caso de estar disponible el libro será de color verde (*success*) y dirá “Disponible”, y sino será rojo con la palabra “Reservado”.

El signo de pregunta en *rating* refiere a que la propiedad *length* no va a ser accedida, en caso de que el valor recibido sea *nulo*.

Nuestra página se debería ver actualmente así:

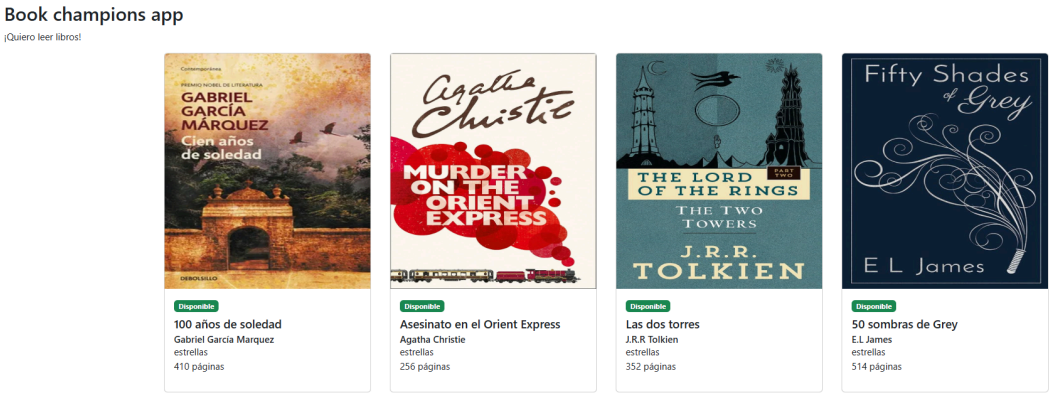


Ejercicio 1

Vamos a crear un componente que engloba a todos los "BookItem" dentro de "App". El componente se llamará "Books" y de esa manera, "App" nos va a quedar más prolijo y mejor segmentado.

Pasos:

1. Crear el componente "Books", que retornara los distintos "BooksItems".
2. Mantener la data de *books* en el componente "App" y pasarla por *props* al componente "Books".
3. Aplicar las siguientes clases de *bootstrap* al *div* principal de Books *className="d-flex justify-content-center flex-wrap"*
4. Luego, en BookItem aplicar la siguiente clase al componente Card *className="mx-3"*



Composición en React

¿A qué nos referimos con el concepto de composición en React? La composición la efectuamos en el momento en que utilizamos a un componente como "cáscara" de otros componentes, en general, para aplicar los mismos estilos a los componentes que se encuentran encerrados.

Pensemos en un componente llamado por ejemplo "Layout", que es el *layout* de las pantallas de nuestro sistema web. Es muy probable que queramos que todas las pantallas posean el mismo estilo y los mismos componentes compartidos (por ejemplo, que en todas las pantallas del sistema se vea la barra de navegación superior).

React toma los valores que encierran sus etiquetas abiertas y se lo asigna a la *prop* llamada *children* (que viene por defecto en todos los componentes). Por ejemplo, si yo quisiera crear un botón de cero como etiqueta abierta:

<MyButton> Comprar Libro</MyButton>

<MyButton> Iniciar Sesión </MyButton>

<MyButton>Volver a menú principal</MyButton>

Donde el componente MyButton comparta estilizado, funcionalidades y que el texto lo reciba según lo que el desarrollador necesite en el momento.

Para ello, la declarativa de mi componente podría ser:

const MyButton = ({children}) => {

```
    return <button>{children}</button>
};

export default MyButton;
```

Donde children sería igual en cada caso a los textos:

“Comprar Libro”
“Iniciar Sesión”
“Volver al menú principal”

Esto no solo se reduce a texto o valores, sino que también *children* puede representar fragmentos de código de React, como por ejemplo el componente de *bootstrap* Card, recibe como *children*:

```
<Card.Body>
    <div className="mb-2">
        {available ?
            <Badge bg="success">Disponible</Badge>
            :
            <Badge bg="danger">Reservado</Badge>
        }
    </div>
    <Card.Title>{title}</Card.Title>
    <Card.Subtitle>{author}</Card.Subtitle>
    <div>{rating} estrella{rating > 1 ? 's' : ''}</div>
    <p>{pageCount} páginas</p>
    <Button >
        Actualizar titulo
    </Button>
</Card.Body>
```

Nota: en el caso de MyButton, ¿Podríamos envíar el texto como *props*, del estilo <MyButton text=“Comprar Libro” />? Si, totalmente. Pero sucede que en algunos casos (como en el que encerramos fragmentos de código React o en ejemplos muy familiares de HTML como *button*) es más normal o cotidiano trabajar con etiquetas abiertas que cerradas, principalmente en casos de desarrollo que vienen con una fuerte base HTML.

Fecha	**Versionado actual**	**Autor**	**Observaciones**
28/03/2022	1.0.0	Gabriel Golzman	Primera versión
30/07/2022	1.1.0	Gabriel Golzman	Actualización nueva versión *index.js*
19/01/2023	1.1.1	Gabriel Golzman	Actualizado imágenes nuevo *bg*
23/11/2023	2.0.0	Gabriel Golzman	Segunda versión
03/12/2024	3.0.0	Gabriel Golzman	Tercera versión