

Universidad Tecnológica Nacional
Facultad Regional Rosario
Tecnicatura Universitaria en Programación



Programación III
Unidad 4.3
Contexto y *custom hooks*

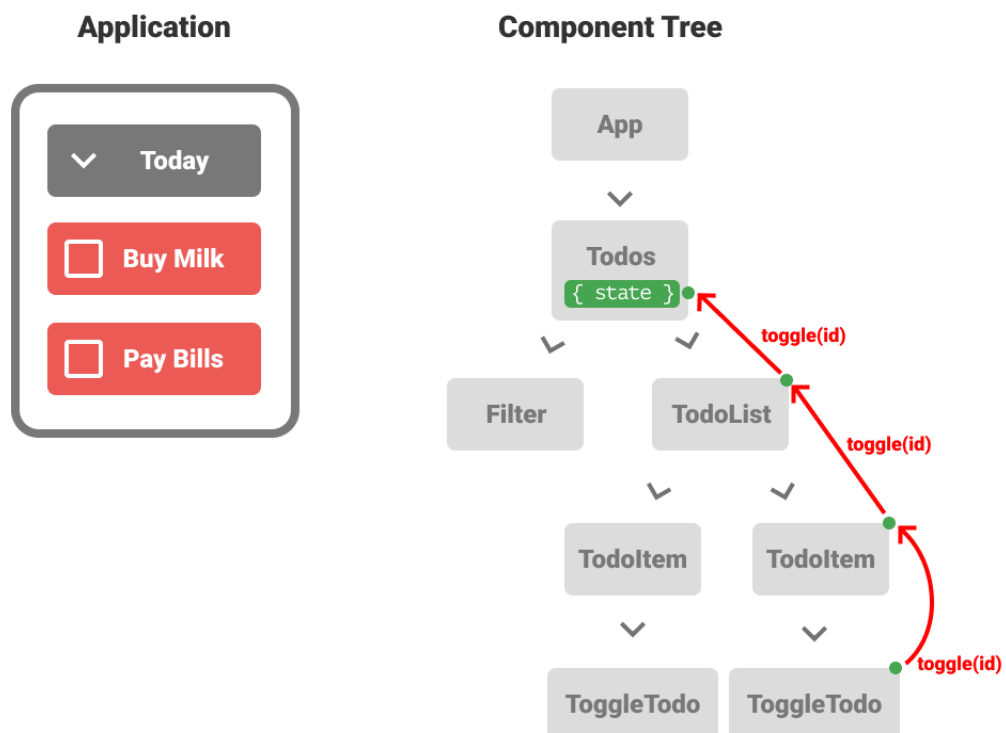
Contexto en React	3
Introducción	3
La store como solución	4
Múltiples contextos	4
Authentication Context	5
main.jsx	6
App	6
Login	7
Protected	8
Theme Context	9
ToggleTheme	10
Hooks personalizados	12
Ejemplo aplicado a books-champion: useTranslate	12

Contexto en React

Introducción

Hemos visto que para poder pasar valores entre componentes solo lo podemos hacer de manera vertical, sea hacia abajo mediante la utilización de *props* o hacia arriba utilizando una *callback* para subir los valores al componente padre.

Si esos valores deseamos pasarlos a través de varios nodos, es cuestión solo de atravesar esos componentes “de paso” hasta llegar al componente deseado. Esta acción, además de poco práctica a nivel de consistencia de código, nos lleva a tener una aplicación que sufre del *props drilling*, donde ciertos valores tengan que movilizarse a través de todos los componentes (pensemos en la opción de tema claro o tema oscuro para la app, donde todos los componentes de nuestra aplicación cambian su color teniendo en cuenta esta opción)



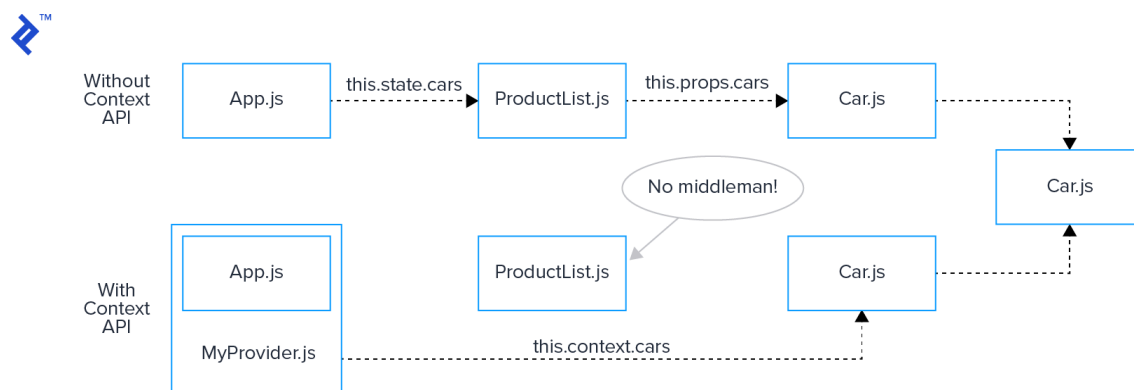
Ejemplo de props drilling para subir el valor de la id desde ToggleTodo a Todos

Además, recordemos que el pasaje de valores de manera horizontal no es posible en React, para ello solo podemos subir el valor hasta un componente padre en común y luego bajar por *props* dicho valor.

La *store* como solución

Librerías como Redux (y, el contexto en React) nos traen la solución a estos problemas mediante la implementación de una **store** (tienda) en donde guardamos todos los valores que deseemos que accedan los componentes (los componentes seleccionados que tienen permitido acceder a estos valores serán determinados por el *provider* (proveedor)).

Estos valores pueden incluso ser funciones que realizan modificaciones deseadas.



En la imagen anterior, vemos la diferencia utilizando *Context* en React. Sin *Context*, debemos pasar la lista de autos desde que se obtiene en App hasta llegar a Car.js.

Luego con la utilización de un provider que encierre a App, el Car Context ya le proporciona a Car la lista de autos sin la necesidad de pasar por el componente Product

Múltiples contextos

React nos permite crear múltiples contextos que convivan según las necesidades de la aplicación. A continuación, armaremos 2 distintas:

- **AuthenticationContext:** en donde el usuario al hacer login, guardaremos el token.
- **ThemeContext:** en donde mediante un *swich* en la UI, intercambie el tema de la aplicación.

Authentication Context

Para poder crear un contexto nuevo, crearemos la carpeta *services* dentro de *components* y creamos la carpeta *auth*, que contendrá el archivo *auth.context.jsx* (allí crearemos y exportaremos el contexto) y el archivo *AuthProvider.jsx* (componente proveedor de dicho contexto).

```
1  import { createContext } from "react";
2
3  export const AuthenticationContext = createContext();
```

```
1  import { useState } from "react";
2  import { AuthenticationContext } from "../auth.context";
3
4  const tokenValue = localStorage.getItem("book-champions-token");
5
6  export const AuthenticationContextProvider = ({ children }) => {
7    const [token, setToken] = useState(tokenValue);
8
9    const handleUserLogin = (token) => {
10     localStorage.setItem("book-champions-token", token);
11     setToken(token);
12   };
13
14   const handleUserLogout = () => {
15     localStorage.removeItem("book-champions-token");
16     setToken(null);
17   };
18
19   return (
20     <AuthenticationContext value={{ token, handleUserLogin, handleUserLogout }}>
21       {children}
22     </AuthenticationContext>
23   );
24 };
25
26
27
```

Nuestro objetivo principalmente es guardar el token en el contexto, de manera de que todos los componentes que deseen acceder a ese valor, puedan hacerlo.

En este contexto:

- Declaramos por fuera del proveedor un *tokenValue* que tomará el valor de lo que tengamos guardado en el *localStorage* con la key "book-champions-token".
- Nuestro proveedor posee un estado que comienza con ese valor obtenido y formateado del *localStorage*.
- Luego, en el *handleUserLogin* tomamos el parámetro del email y seteamos tanto el estado como el *local storage*.
- En el *handleUserLogout*, quitamos el ítem del *local storage* y seteamos el estado en *null*.

El proveedor retorna en su `jsx` entonces al contexto para que arme la *store* con los valores que le pasamos en `value` (*value* debe ser si o si un objeto).

Finalmente con el concepto de composición devolvemos lo que encierra este `jsx`.

Ahora debemos realizar múltiples cambios en nuestro código para que refleje esta nueva sintaxis.

¿A qué componentes les puede interesar esa información?

Un caso de uso posible sería en el componente `Protected`, el cual redirige al usuario si no está logueado a `/login`, y sino devuelve la ruta solicitada. Allí debemos comprobar que exista el token y **además, comprobar que no haya expirado** (recordemos que el token tenía 1 hora de vigencia)

`main.jsx`

```
9   createRoot(document.getElementById('root')).render(  
10     <StrictMode>  
11       <AuthenticationContextProvider>  
12         <App />  
13       </AuthenticationContextProvider>  
14     </StrictMode>,  
15   )
```

Importamos y encerramos `App` en nuestro proveedor de autenticación, para que todos sus componentes puedan acceder a la *store*.

`App`

Borraremos todo lo que tenga que ver con nuestra lógica vieja de `Login`:

```

12  const App = () => {
13    return (
14      <div className="d-flex flex-column align-items-center">
15        <ToastContainer />
16        <BrowserRouter>
17          <Routes>
18            <Route path="/" element={<Navigate to='login' />} />
19            <Route path="/login" element={<Login />} />
20            <Route path="/register" element={<Register />} />
21            <Route element={<Protected />}>
22              <Route
23                path="/library/*"
24                element={
25                  <Dashboard />
26                }>
27              </Route>
28            </Route>
29            <Route path="*" element={<NotFound />} />
30          </Routes>
31        </BrowserRouter>
32      </div>
33    )
34  }

```

Login

```

21  const { handleUserLogin } = useContext(AuthenticationContext)
22
54  loginUser(
55    email,
56    password,
57    token => {
58      handleUserLogin(token)
59      navigate("/library");
60    },
61    err => {
62      errorToast(err.message)
63    }
64  )
65  }

```

Cambiaremos la función `onLogin` por nuestro *handleLogin* del Authentication Provider. La forma que se muestra más arriba, es la forma de acceder al contexto, **utilizando el hook `useContext` y desestructurando sus propiedades**.

Protected

```

1 | import { useContext } from "react";
2 | import { Navigate, Outlet } from "react-router";
3 |
4 | import { AuthenticationContext } from "../../services/auth/auth.context";
5 | import { isValidToken } from "../auth.helpers";
6 |
7 | const Protected = () => {
8 |   const { token } = useContext(AuthenticationContext);
9 |   if (!isValidToken(token)) {
10 |     return <Navigate to="/login" replace />;
11 |   }
12 |
13 |   return <Outlet />;
14 |
15 | };
16 |
17 | export default Protected;
```

En Protected, sencillamente veremos si hay un token seteado válido, si no lo redirigiremos al Login.

```

34 | export const isValidToken = (token) => {
35 |   if (!token) return false;
36 |   try {
37 |     const decodedToken = jwtDecode(token);
38 |
39 |     // We need to convert Date.now() from milliseconds to seconds
40 |     const currentTime = Date.now() / 1000;
41 |
42 |     return currentTime < decodedToken.exp;
43 |   } catch (error) {
44 |     console.error('Error decoding token:', error);
45 |     return false;
46 |   }
47 | }
```

Nota: instalar la librería *jwt-decode* para que este método funcione correctamente. Podemos probar la expiración del token cambiando el tiempo de expiración en 60 segundos en Node.


```

50     const token = jwt.sign({ email }, secretKey, { expiresIn: 60 });
51

```

Luego en Dashboard, hacemos un trabajo similar para “Cerrar sesión” con el *handleLogout*.

Theme Context

Crearemos la carpeta theme dentro de services y luego el archivo theme.context.jsx y ThemeContextProvider.jsx.

```

1  import { useEffect, useState } from "react";
2
3  import { ThemeContext } from "../theme.context";
4  import { DARK_THEME, LIGHT_THEME } from "../consts";
5
6  const themeValue = localStorage.getItem('theme');
7
8  const ThemeContextProvider = ({ children }) => {
9      const [theme, setTheme] = useState(themeValue);
10
11      useEffect(() => {
12          document.documentElement.setAttribute("data-bs-theme", themeValue);
13      }, []);
14
15      const toggleTheme = () => {
16          if (theme === LIGHT_THEME) {
17              document.documentElement.setAttribute("data-bs-theme", DARK_THEME);
18              localStorage.setItem("theme", DARK_THEME);
19              setTheme(DARK_THEME);
20          } else {
21              document.documentElement.setAttribute("data-bs-theme", LIGHT_THEME);
22              localStorage.setItem("theme", LIGHT_THEME);
23              setTheme(LIGHT_THEME);
24          }
25      };
26
27
28      return (
29          <ThemeContext value={{ theme, toggleTheme }}>
30              {children}
31          </ThemeContext>
32      );
33  }
34
35  export default ThemeContextProvider;

```

La *store* será entonces el valor de theme (“light”, “dark”) y la función toggle, que permite cambiarlo.

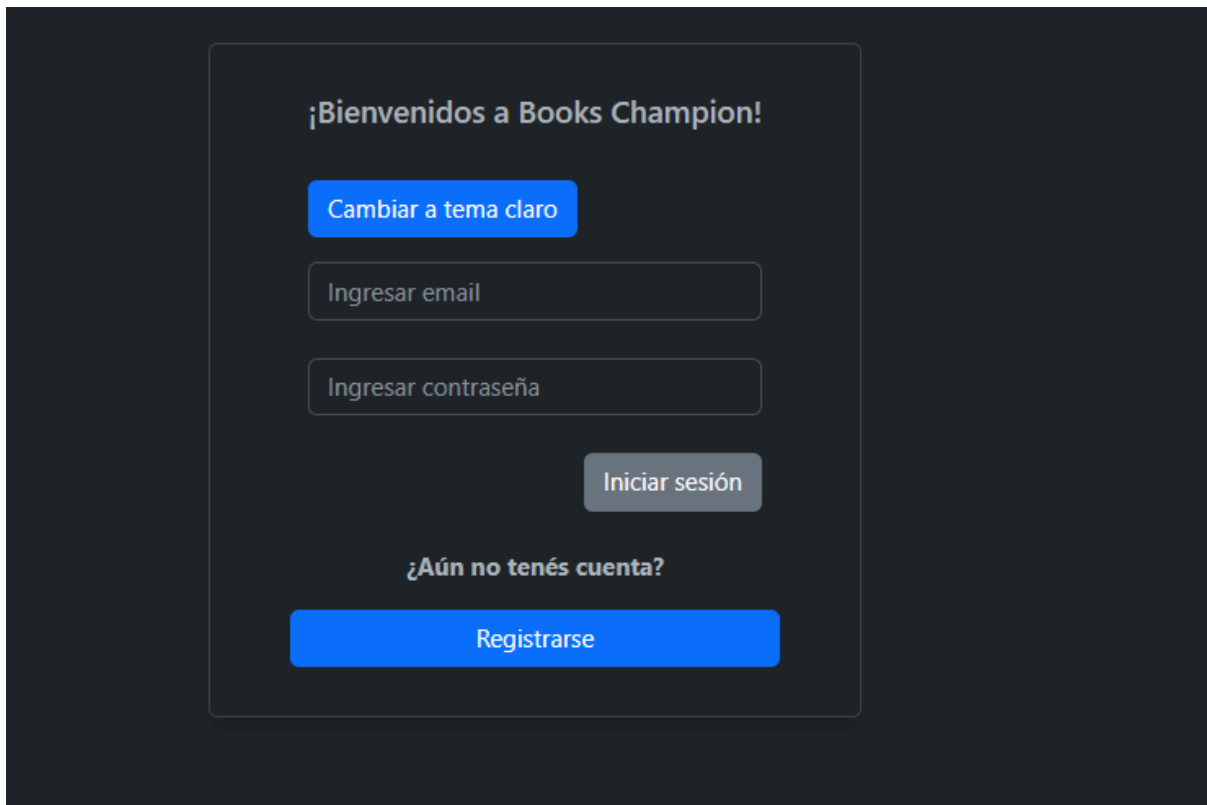
Luego, agregaremos el proveedor también a index.js.

ToggleTheme

Crearemos el componente ToggleTheme, que nos permitirá cambiar el tema de la aplicación mediante un botón:

```
1  import { useContext } from "react";
2  import { Button } from "react-bootstrap";
3
4  import { ThemeContext } from "../../../../../services/theme/theme.context";
5  import { LIGHT_THEME } from "../../../../../services/consts";
6
7  const ToggleTheme = () => {
8    const { theme, toggleTheme } = useContext(ThemeContext);
9
10    return (
11      <Button onClick={toggleTheme} className="me-3 my-3">
12        Cambiar a tema {theme === LIGHT_THEME ? "oscuro" : "claro"}
13      </Button>
14    );
15  };
16
17  export default ToggleTheme;
```

Luego agregamos ToggleTheme al Login, Register y al Dashboard:



¡Bienvenidos a Books Champion!

Cambiar a tema claro

Ingresar nombre de usuario

Ingresar email

Ingresar contraseña

Iniciar sesión Registrarse

Cambiar a tema claro Agregar libro Cerrar sesión

Book champions app

¡Quiero leer libros!

Buscar libro...

Hooks personalizados

Los hooks personalizados de React nos dan la capacidad de extraer y reusar la lógica de ciertos componentes. Pensemos en la arquitectura de componentes, donde se reutiliza el maquetado, estilizado y lógica, pero en los *hooks* personalizados nos enfocamos solo en la lógica.

Es básicamente entonces una función de JavaScript que comienza con el prefijo *use* para que React la reconozca y nos permita utilizar los otros hooks de React (*useState*, *useEffect*, *useCallback*) dentro suyo, sin estar dentro de un componente funcional.

Ejemplo aplicado a *books-champion*: *useTranslate*

Supongamos que deseamos crear un hook que nos permita hacer un *switch* entre varios idiomas para los textos de nuestra web. Una posible aplicación podría ser algo del estilo:

```
1  import { useContext } from "react";
2
3  import { TranslationContext } from "../../services/translation/translation.context";
4  import { translation_dictionary } from "../translation.dictionary";
5
6  export const useTranslate = () => {
7      const { language } = useContext(TranslationContext);
8
9      return (key) => {
10         const translation = translation_dictionary[language]
11             ? translation_dictionary[language].find((t) => t.key === key)?.value
12             : translation_dictionary["en"].find((t) => t.key === key)?.value;
13
14         return translation || key;
15     };
16 };
17
```

Donde en el contexto guardamos el lenguaje seleccionado por el usuario:

```
1  import { useState } from "react";
2
3  import { TranslationContext } from "../translation.context";
4
5  const tValue = localStorage.getItem("translation");
6
7  const TranslateContextProvider = ({ children }) => {
8    const [language, setLanguage] = useState(tValue ?? "en");
9
10    const changeLanguageHandler = (newLanguage) => {
11      localStorage.setItem("translation", newLanguage);
12      setLanguage(newLanguage);
13    };
14    return (
15      <TranslationContext value={{ language, changeLanguageHandler }}>
16        {children}
17      </TranslationContext>
18    );
19  };
20
21  export default TranslateContextProvider;
22
```

Luego, *useTranslate* nos devuelve una función que toma como parámetro la key para realizar la búsqueda dentro de un diccionario y devolver, según esa key y el lenguaje seleccionado, la traducción correspondiente:

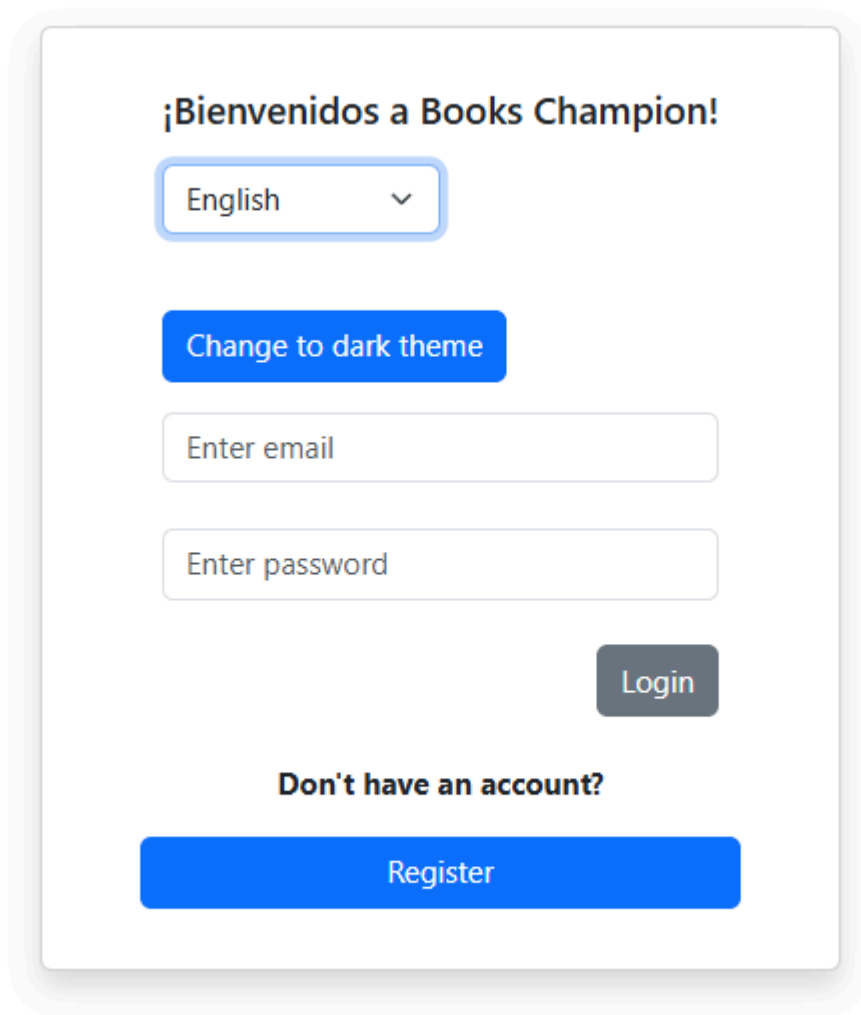
```

1  export const translation_dictionary = {
2    es: [
3      { key: "login", value: "Iniciar sesión" },
4      { key: "dark_theme_change", value: "Cambiar a tema oscuro" },
5      { key: "light_theme_change", value: "Cambiar a tema claro" },
6      { key: "welcome", value: "¡Bienvenidos a Books Champion!" },
7      { key: "spanish_lang", value: "Español" },
8      { key: "english_lang", value: "Inglés" },
9      { key: "username", value: "nombre de usuario" },
10     { key: "email", value: "email" },
11     { key: "password", value: "Contraseña" },
12     { key: "enter", value: "Ingresar" },
13     { key: "incorrect", value: "incorrecto" },
14     { key: "login_no_account", value: "¿Aún no tenés cuenta?" },
15     { key: "register", value: "Registrarse" },
16     { key: "username_empty", value: "Debe ingresar un nombre de usuario" },
17     { key: "email_empty", value: "Debe ingresar un email" },
18     { key: "password_empty", value: "Debe ingresar una contraseña" },
19   ],
20   en: [
21     { key: "login", value: "Login" },
22     { key: "dark_theme_change", value: "Change to dark theme" },
23     { key: "light_theme_change", value: "Change to light theme" },
24     { key: "welcome", value: "Welcome to Books Champion!" },
25     { key: "spanish_lang", value: "Spanish" },
26     { key: "english_lang", value: "English" },
27     { key: "username", value: "username" },
28     { key: "email", value: "email" },
29     { key: "password", value: "password" },
30     { key: "enter", value: "Enter" },
31     { key: "incorrect", value: "incorrect" },
32     { key: "login_no_account", value: "Don't have an account?" },
33     { key: "register", value: "Register" },
34     { key: "username_empty", value: "You must enter a username" },
35     { key: "email_empty", value: "You must enter an email" },
36     { key: "password_empty", value: "You must enter a password" },
37   ],
38 },
39 };

```

Para la aplicación de este traductor, crearemos un *dropdown* o Combo para seleccionar mediante UI el lenguaje:

```
1 import { useContext } from "react";
2 import { Form } from "react-bootstrap";
3
4 import { TranslationContext } from "../../../../../services/translation/translation.context";
5 import { useTranslate } from "../../../../../custom/useTranslate/useTranslate";
6
7 const ComboLanguage = () => {
8   const { language, changeLanguageHandler } = useContext(TranslationContext);
9
10  const translate = useTranslate();
11
12  const changeLanguage = (event) => {
13    changeLanguageHandler(event.target.value);
14  };
15
16  return (
17    <Form.Select
18      onChange={changeLanguage}
19      value={language}
20      aria-label="Select Language"
21      className="w-50 mb-4"
22    >
23      <option value="es">{translate("spanish_lang")}</option>
24      <option value="en">{translate("english_lang")}</option>
25    </Form.Select>
26  );
27 };
28
29 export default ComboLanguage;
```



¡Bienvenidos a Books Champion!

English ▼

Change to dark theme

Enter email

Enter password

Login

Don't have an account?

Register

Ese combo (que ubicamos dentro de Login) nos permite realizar el *switch* entre el idioma español y el inglés. Podemos, como ejercicio de práctica, aplicar la misma lógica para traducir todo el dashboard.

Los cambios a realizar son:

1. Importamos la función que nos devuelve el *hook*

```
27  
28 const t = useTranslate();  
29
```

2. Con las *keys*, vamos traduciendo los textos visibles para el usuario (por ejemplo, el *input* de email):


```
<FormGroup className="mb-4">
  <Form.Control
    autoComplete="email"
    type="email"
    className={errors.email && "border border-danger"}
    ref={emailRef}
    placeholder={` ${t("enter")} ${t("email")}`}
    onChange={handleEmailChange}
    value={email} />
    {errors.email && <p className="mt-2 text-danger">{t("email_empty")}</p>}
</FormGroup>
```

Fecha	Versionado actual	Autor	Observaciones
03/06/2023	1.0.0	Gabriel Golzman	Primera versión
17/01/2024	2.0.0	Gabriel Golzman	Segunda versión
22/02/2025	2.1.0	Gabriel Golzman	Mejoras generales