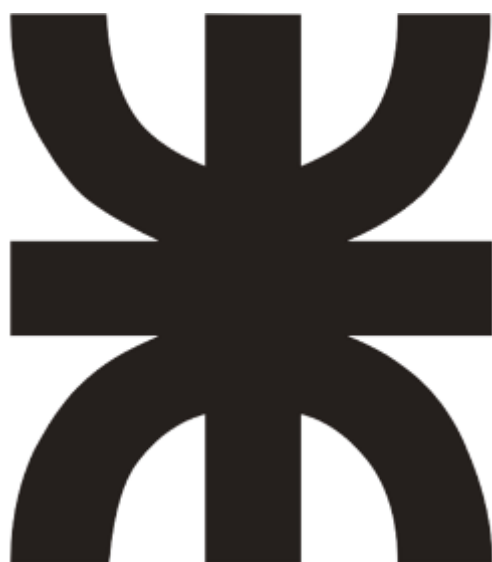


Universidad Tecnológica Nacional
Facultad Regional Rosario
Tecnicatura Universitaria en Programación



Programación III

Unidad 4.2

Autorizaciones

Introducción	3
¿A qué nos referimos con autorización?	3
JWT (json web token)	3
El proceso	4
Contraseñas y encriptación	5
hash	5
Salting	6
Modificando book-champions	6
Creando el schema User	6
Registrar un usuario	8
Prácticas comunes en el registro	9
Iniciar sesión como usuario	9
Volviendo al lado cliente	10
Desafío de clase	10
Registrando e iniciando sesión del lado cliente	12
Middleware y comprobación del token	13
Lado cliente	15
Validaciones	16
Lado servidor	16
Rutas de autorización (register / login)	16
Desafío de clase	18
Lado cliente	18
Mejoras	19
Manejo de errores	19
Separaciones en archivos adicionales	20
Variables de entorno	21

Introducción

¿A qué nos referimos con autorización?

En el desarrollo de sistemas web, la **autorización de usuarios** se refiere comúnmente al permiso que otorgamos a los mismos a acceder a los recursos del sistema, a través de una interfaz de registro / inicio de sesión (*register / login*). Esta autenticación en general se hace a través del ingreso mediante formularios de un nombre de usuario o email, y una contraseña.

Desde hace unos años ya, se ha implementado a su vez el concepto de **autenticación de doble factor**, en donde el usuario además de ingresar sus datos debe completar alguna comprobación adicional, como por ejemplo:

- Código vía SMS o mensaje de whatsapp
- Código vía llamada telefónica
- Autenticación con aplicaciones afines (Google auth, Microsoft Authenticator)
- Huella digital
- Reconocimiento facial

Luego, una vez aprobada su ingreso al sistema, todos los pedidos al servidor que realice el usuario en rutas privadas (es decir, dentro de la web) tendrán que tener información extra para marcarlos como efectivamente autenticados. Nosotros en esta cátedra utilizaremos la tecnología JWT (*JSON web token*).

JWT (*json web token*)

JSON Web Token (JWT) es un protocolo estándar abierto (RFC 7519) que define una forma compacta y auto contenida para transmitir de manera segura, información entre partes como un objeto JSON. Esta información puede ser verificada y confiable, ya que este objeto está firmado de manera digital.

Está compuesto por

- **header**: contiene metadata sobre el token mismo, por ejemplo el algoritmo que se usó para la firma digital.

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

- **payload**: es lo que contiene las *claims*, que son datos adicionales (típicamente del usuario, como su nombre, email, rol)

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

- **signature**: la firma digital se crea combinando el *header* codificado, el *payload* codificado, un secreto (un *string* que se encuentre solo en el servidor) y el algoritmo especificado en el server.

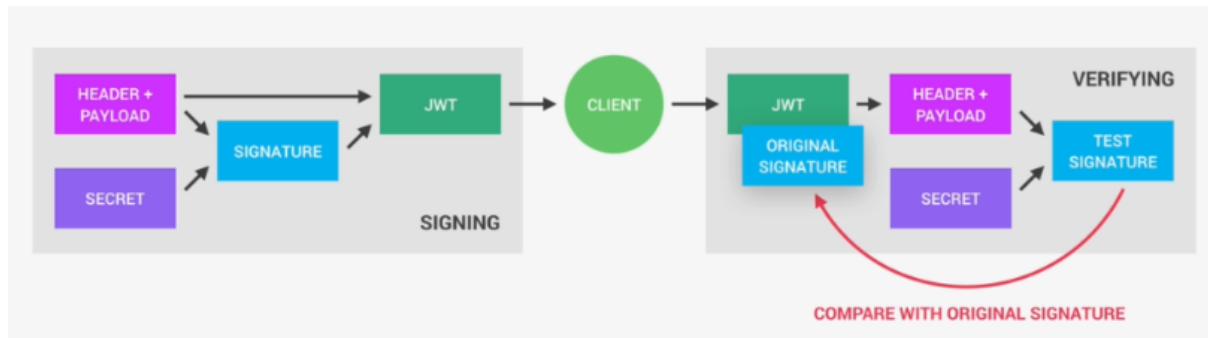
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

El proceso

El servidor se encarga de crear y firmar digitalmente el token, para luego enviarlo al cliente.



Luego, en cada una de las subsiguientes *requests*, el servidor va a verificar que el *header + payload* no haya sido modificado por un agente externo. Para ello, crea una “firma de prueba” a partir del *header + payload* recibido y la compara con la firma original, que se encuentra en el token original. Si las firmas coinciden, el token es válido.



Contraseñas y encriptación

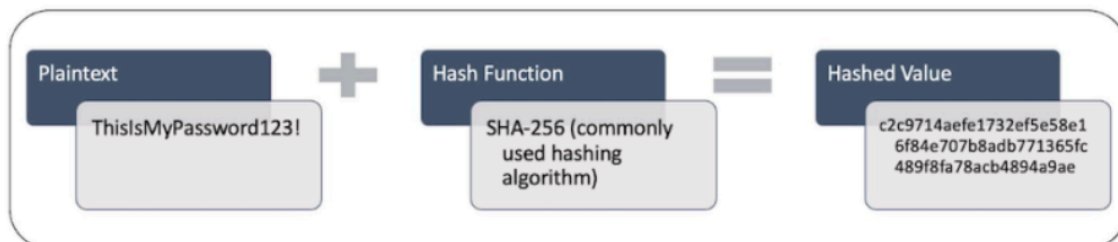
Las contraseñas de usuario las vamos a encriptar mediante métodos de *hash* y *salt*.

hash

El *hash* de una contraseña, es la salida de un algoritmo encargado de modificar el *string* original. Idealmente, ese *hash* obtenido (conjunto de caracteres) es

- Difícil de revertir a su valor original.
- Ser sencillo de calcular basado en el *input*.
- Que posea poca colisión (es decir, contraseñas similares igual deben obtener un *hash* diferente)
- El *hash* debe tener una longitud predeterminada o un rango fijo de longitudes.

El algoritmo más utilizado es el de SHA-256.

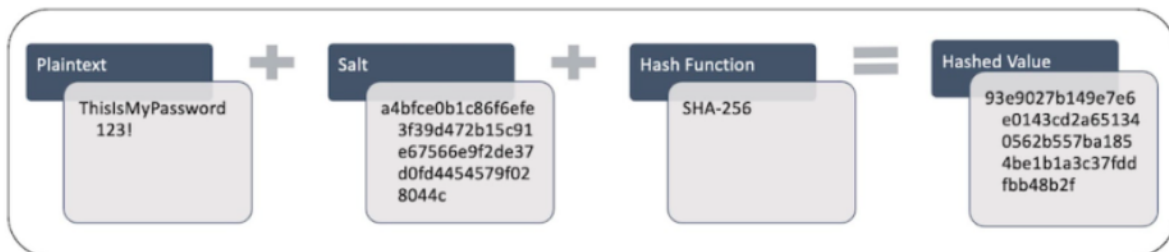


A medida que el poder de cómputo y el espacio en disco aumentó, este tipo de algoritmos poseía una grave falla. Nada impedía que un usuario armara una tabla enorme con los passwords más comunes y sus respectivos *hashes*, e hiciera la comparación ahí dentro. Una vez que tuviera un *match*, podría hackear y obtener todo lo referido al usuario (de ahí a que se nos pida crear contraseñas complejas, de manera que evitemos estos casos comunes y tampoco lo puedan sacar por búsqueda de fuerza bruta)

Para ello, se creó el *salting*.

Salting

Salting es una práctica donde agregamos “ruido” al *hash* original, de manera que sea más difícil de lograr encontrar ese *hash* en una tabla. La librería que vamos a usar, bcrypt, realiza un salting aleatorio distinto **para cada uno de las contraseñas de los usuarios**.



Modificando *book-champions*

Creando el *schema* User

Agregaremos un simple esquema de usuario, con un nombre, email y una contraseña.



```

1  import { DataTypes } from "sequelize";
2  import { sequelize } from "../db";
3
4  export const User = sequelize.define("user", {
5      id: {
6          type: DataTypes.INTEGER,
7          primaryKey: true,
8          autoIncrement: true,
9      },
10     name: {
11         type: DataTypes.STRING,
12         allowNull: true,
13     },
14     email: {
15         type: DataTypes.STRING,
16         allowNull: false
17     },
18     password: {
19         type: DataTypes.TEXT,
20         allowNull: false,
21     }
22 }, {
23     timestamps: false
24 });
25

```

Luego, las rutas de autenticación, dentro de *auth.routes.js*

```

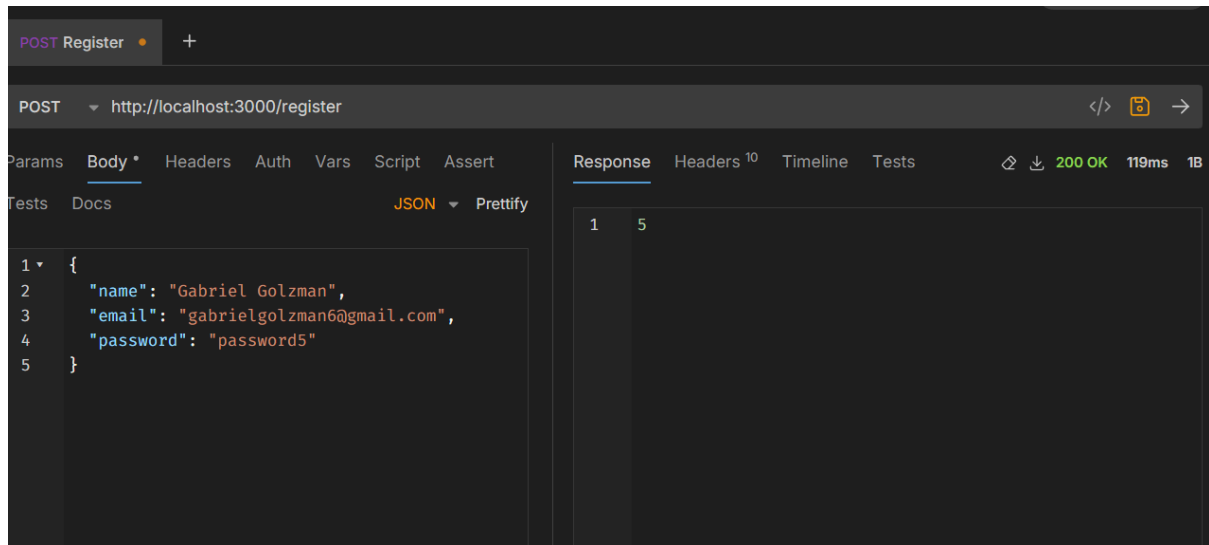
src > routes >  auth.routes.js >  default
1  import { Router } from "express";
2
3  const router = Router();
4
5  router.post("/register", registerUser);
6
7  router.post("/login", loginUser);
8
9  export default router;
10
11

```

Registrar un usuario

Para registrar un usuario, tomaremos el nombre, su correo y el password deseado. Luego, haremos el *hash* de la contraseña para crear al nuevo usuario:

```
7   export const registerUser = async (req, res) => {
8     const { name, email, password } = req.body;
9
10    const user = await User.findOne({
11      where: {
12        email
13      }
14    });
15
16    if (user)
17      return res.status(400).send({ message: "Usuario existente" });
18
19    // Hash the password
20    const saltRounds = 10;
21
22    const salt = await bcrypt.genSalt(saltRounds);
23
24    const hashedPassword = await bcrypt.hash(password, salt);
25
26    const newUser = await User.create({
27      name,
28      email,
29      password: hashedPassword,
30    });
31
32    res.json(newUser.id);
33
34  }
```

Comprobamos también que si el usuario, si ya existe, no nos permite crearlo.

Prácticas comunes en el registro

Es importante mencionar que hay diversas formas de manejar el registro de un usuario. En general, luego del registro del mismo se envía un email con un link de verificación, o se solicita el ingreso de un código también enviado por correo electrónico.

En nuestro caso, ya que el manejo de *emailing* escapa al alcance de la cátedra, sencillamente redirigiremos al usuario a la pantalla de *login* para que ingrese sus datos y se pueda autenticar.

Iniciar sesión como usuario

Recordemos que para la gestión del JSON web token, debemos instalar la librería de node correspondiente:

```
npm i jsonwebtoken
```

Para el inicio de sesión de usuario, compararemos con *bcrypt* la contraseña ingresada y el hash guardado. Luego, generamos el JWT que le permitirá realizar los subsecuentes pedidos al servidor:

```
6 export const loginUser = async (req, res) => {
7   const { email, password } = req.body;
8
9   const user = await User.findOne({
10     where: {
11       email
12     }
13   });
14
15   if (!user)
16     return res.status(401).send({ message: "Usuario no existente" });
17
18   const comparison = await bcrypt.compare(password, user.password);
19
20   if (!comparison)
21     return res.status(401).send({ message: "Email y/o contraseña incorrecta" });
22
23   // Generate token
24   const secretKey = 'programacion3-2025';
25
26   const token = jwt.sign({ email }, secretKey, { expiresIn: '1h' });
27
28   return res.json(token);
29 }
```

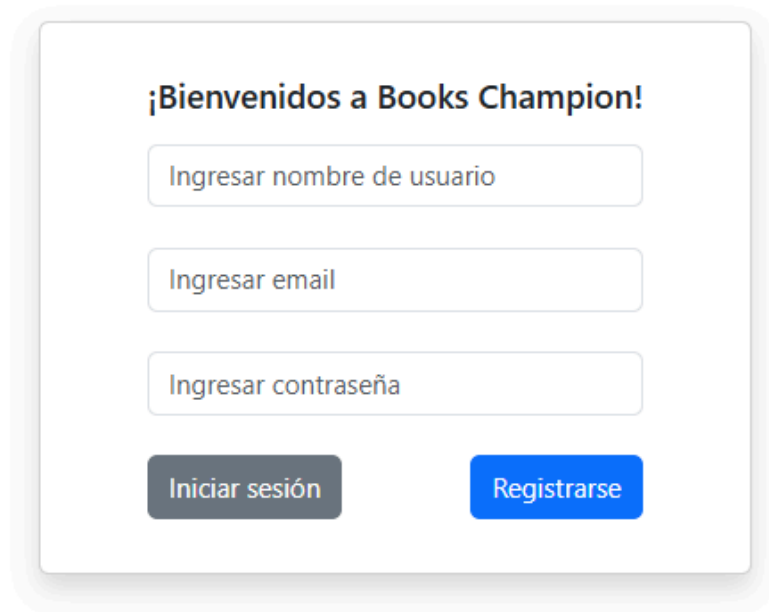
La generación de token sucede a partir del email del usuario, de una *secret key* (la misma, luego la movemos de ahí por razones de seguridad) y la configuración (en nuestro caso, la expiración del token)

Volviendo al lado cliente

Desafío de clase

Debemos crear una ruta muy similar al login, donde el usuario pueda registrarse (con los campos nombre, correo electrónico y contraseña). Luego, manejar los estados y la llamada al servidor de manera que efectivamente podamos registrar un usuario.

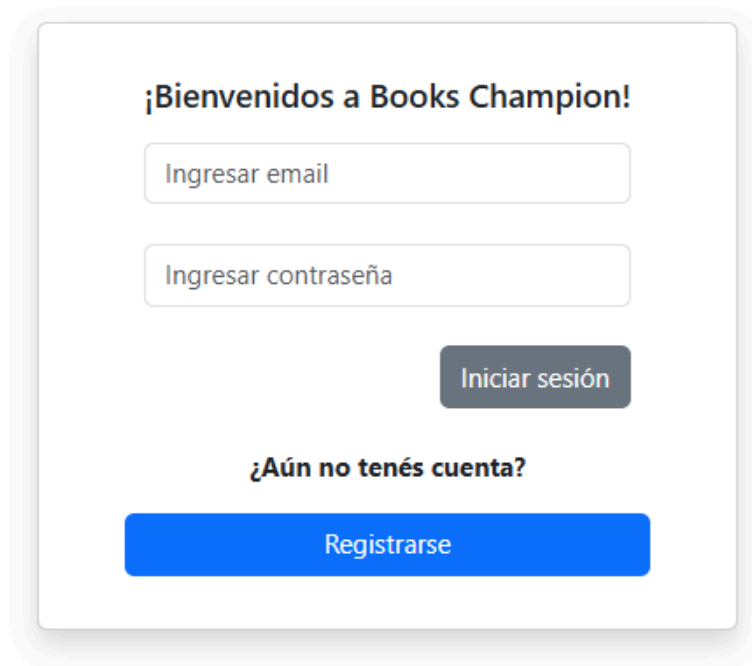
Así nos quedaría el componente de Register:



¡Bienvenidos a Books Champion!

Formulario de registro con tres campos de entrada: 'Ingresar nombre de usuario', 'Ingresar email', y 'Ingresar contraseña'. Debajo de los campos hay dos botones: 'Iniciar sesión' (gris) y 'Registrarse' (azul).

Y así el de Login, mejorado:



¡Bienvenidos a Books Champion!

Formulario de login con dos campos de entrada: 'Ingresar email' y 'Ingresar contraseña'. Debajo de los campos hay un botón 'Iniciar sesión' (gris). En la parte inferior, hay un texto '¿Aún no tenés cuenta?' y un botón 'Registrarse' (azul).

Como vemos, ambos comparten *background* y *header*. Si es posible, crear un nuevo componente AuthContainer, donde el contenido del formulario ocupe la posición de la *prop children*, de esta manera:

```

6   const Login = ({ onLogin }) => {
55   return (
56     <AuthContainer>
57       <Form onSubmit={handleLogin}>
58         <FormGroup className="mb-4">
59           <Form.Control
60             autoComplete="email"
61             type="email"
62             className={errors.email && "border border-danger"}
63             ref={emailRef}
64             placeholder="Ingresar email"
65             onChange={handleEmailChange}
66             value={email} />
67           {errors.email && <p className="mt-2 text-danger">Debe ingresar un email</p>}
68         </FormGroup>
69         <FormGroup className="mb-4">
70           <Form.Control
71             autoComplete="current-password"
72             type="password"
73             className={errors.password && "border border-danger"}
74             ref={passwordRef}
75             placeholder="Ingresar contraseña"
76             onChange={handlePasswordChange}
77             value={password}
78           />
79           {errors.password && <p className="mt-2 text-danger">Debe ingresar un password</p>}
80         </FormGroup>
81         <Row>
82           <Col />
83           <Col md={6} className="d-flex justify-content-end">
84             <Button variant="secondary" type="submit">
85               Iniciar sesión
86             </Button>
87           </Col>
88         </Row>
89         <Row className="mt-4">
90           <p className="text-center fw-bold">¿Aún no tenés cuenta?</p>
91           <Button onClick={handleRegisterClick}>Registrarse</Button>
92         </Row>
93       </Form>
94     </AuthContainer>

```

Registrando e iniciando sesión del lado cliente

El método fetch correspondiente al registro nos quedaría algo de esta forma:

```

51   fetch("http://localhost:3000/register", {
52     headers: {
53       "Content-type": "application/json"
54     },
55     method: "POST",
56     body: JSON.stringify({ name, email, password })
57   })
58   .then(res => res.json())
59   .then(() => {
60     successToast("¡Usuario creado exitosamente!")
61     navigate("/login");
62   })
63   .catch(err => console.log(err))
64 }

```

Para el inicio de sesión, modificamos el *handleLogin* para que haga la llamada al servidor. Una vez obtenido correctamente el token, lo guardamos en el *local storage* para su uso en las subsiguientes llamadas no públicas de la app.

```

46      setErrors({ email: false, password: false })
47      onLogin();
48      fetch("http://localhost:3000/login", {
49        headers: {
50          "Content-type": "application/json"
51        },
52        method: "POST",
53        body: JSON.stringify({ email, password })
54      })
55        .then(res => res.json())
56        .then(token => {
57          localStorage.setItem("book-champions-token", token)
58          navigate("/library");
59        })
60        .catch(err => console.log(err))
61    }

```

Luego, al cerrar sesión eliminamos ese token del *local storage*.

```

20    const handleLogout = () => {
21      setLoggedIn(false);
22      localStorage.removeItem("book-champions-token");
23    }

```

Nota: si bien poseemos un *catch* para los errores en la llamada al servidor, si intentamos registrar un usuario con un mail existente, **la salida va a ser por el mensaje de éxito**, si bien el usuario no es creado ya que el servidor efectivamente va a devolver un error. El por qué de esto y la solución, la veremos en el apunte **4.3 Mejoras y validaciones**.

Middleware y comprobación del token

Es momento de proteger todas aquellas rutas no públicas del lado servidor. En nuestra app, por ahora solo las que refieren a la entidad libro deben ser resguardadas de solicitud sin token.

Crearemos una función que nos permita verificar ese token:

```

6  export const verifyToken = (req, res, next) => {
7    const header = req.header("Authorization") || "";
8    const token = header.split(" ")[1];
9    if (!token) {
10     return res.status(401).json({ message: "No posee autorización requerida" });
11    }
12    try {
13     const payload = jwt.verify(token, 'programacion3-2025');
14     console.log(payload)
15     next();
16   } catch (error) {
17     return res.status(403).json({ message: "No posee permisos correctos" });
18   }
19 }

```

Allí, obtendremos del *header* Authorization el valor del token, que primero consultaremos si existe y luego si es válido para la sesión. En el caso de ser válido, nos devuelve un *payload* (con la información del token) y utilizamos la función *next*, para que continúe el ciclo de vida de la solicitud.

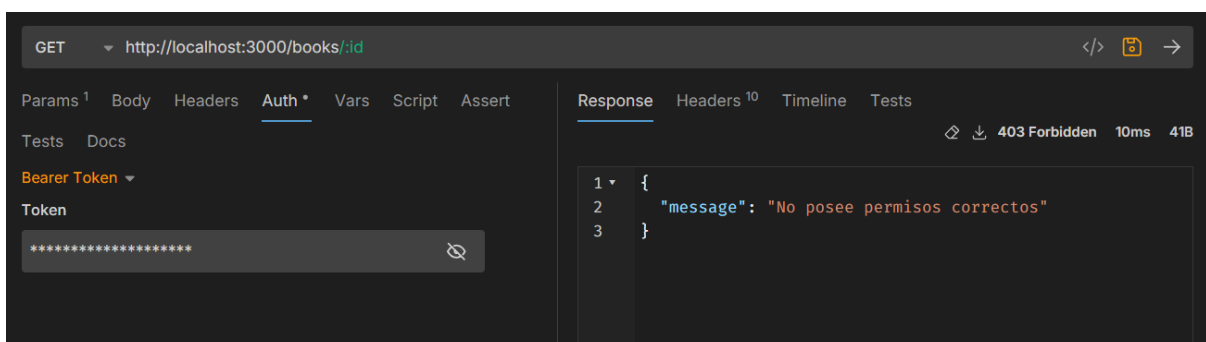
Luego, este middleware lo podemos agregar a cada una de las rutas de libros:

```

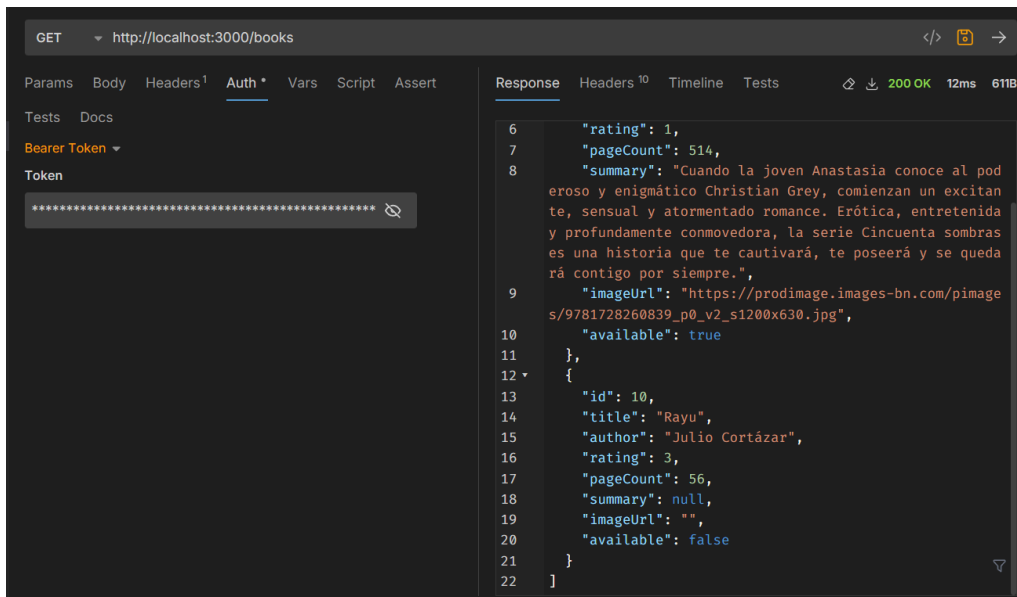
5  const router = Router();
6
7  router.get("/books", verifyToken, findBooks);
8
9  router.get("/books/:id", verifyToken, findBook);
10
11 router.post("/books", verifyToken, createBook);
12
13 router.put("/books/:id", verifyToken, updateBook);
14
15 router.delete("/books/:id", verifyToken, deleteBook);
16

```

Testeamos con el programa Bruno que funcione correctamente. Si no aportamos el token, estas rutas deberían darnos el siguiente mensaje:



Ahora, si hacemos Login y copiamos el token en la pestaña auth de las rutas protegidas, deberíamos poder ver los libros.



Lado cliente

La modificación necesaria en el lado cliente es agregar ese token obtenido (que guardamos en el *local storage*) para realizar las solicitudes:

```

16  useEffect(() => {
17    if (location.pathname === "/library") {
18      fetch("http://localhost:3000/books", {
19        headers: {
20          "Authorization": `Bearer ${localStorage.getItem("book-champions-token")}`
21        }
22      })
23        .then(res => res.json())
24        .then(data => setBookList([...data]))
25        .catch(err => console.log(err));
26    }
27  }, [location])
28
29
    
```

Y con eso deberíamos poder acceder a los recursos, sin problemas.

Validaciones

La validación es un concepto que debemos aplicar en cada una de las capas de nuestro sistema web. Consiste en comprobar que todo aquél dato externo (como el ingreso de datos en un formulario o la llegada de valores a nuestro servidor) sea, justamente, **válido para los requisitos de nuestro negocio**.

Ya que en la capa de base de datos la validación está dada por el mismo motor (que nos dará un error cuando, por ejemplo, el tipo de dato guardado no sea correcto), vamos a concentrarnos en el lado servidor y cliente.

Lado servidor

Buscamos que todos los valores que lleguen por pedido (*request*) sean válidos para los servicios que nosotros escribimos. Por ejemplo, que la contraseña posea valor, tenga más de 7 caracteres de longitud y tenga una mayúscula y un número, al menos.

Una pregunta válida sería ¿Si nosotros verificamos la entrada del usuario por el lado cliente, no sería ya suficiente? Esto no es así, debido a que el usuario puede tratar de obtener información de nuestro servidor **utilizando software de manera similar a como lo usamos nosotros para testear dicho servidor**. Con herramientas como Postman o Bruno podrían llegar a tratar de comunicarse directo al servidor. En líneas generales, es buena práctica la validación en ambas capas.

Rutas de autorización (*register / login*)

Campos a validar:

- **username**: este debe ser alfanumérico, necesario y con un máximo de 13 caracteres.
- **email**: este debe ser un texto del tipo email y es necesario.
- **password**: requerido, con una longitud mayor a 7, una minúscula y una mayúscula.

Creamos un archivo llamado *validations.js*, dentro de una nueva carpeta *helpers*. Allí guardaremos las validaciones más generales, no específicas de una ruta:

```

1  export const validateString = (str, minLength, maxLength) => {
2    if (minLength && str.length < minLength)
3      return false;
4    else if (maxLength && str.length > maxLength)
5      return false;
6
7    return true;
8  }

```



```
10 export const validateEmail = (email) => {  
11   const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;  
12   return emailRegex.test(email);  
13 }
```

```
15 export const validatePassword = (password, minLength, maxLength, needsUppercase, needsNumber) => {  
16   if (minLength && password.length < minLength)  
17     return false;  
18  
19   else if (maxLength && password.length > maxLength)  
20     return false;  
21  
22   else if (needsUppercase && !/[A-Z]/.test(password))  
23     return false;  
24  
25   else if (needsNumber && !/\d/.test(password))  
26     return false;  
27  
28  
29   return true;  
30 }
```

Vemos que en algunos casos, utilizamos el concepto de **expresión regular** para comprobar los *strings*. Una expresión regular no son más que patrones para hacer coincidir combinaciones de caracteres en cadenas. A esta declarativa de patrones, luego se le aplica la función *test* que devuelve verdadero si el patrón se cumple, y sino falso.

Luego, podemos utilizar estas funciones para validar la ruta de login.

```

22
23 export const loginUser = async (req, res) => {
24
25     const result = validateLoginUser(req.body);
26
27     if (result.error)
28         return res.status(400).send({ message: result.message })
29

```

```

91 // Validations
92 const validateLoginUser = (req) => {
93     const result = {
94         error: false,
95         message: ''
96     }
97     const { email, password } = req;
98
99     if (!email || !validateEmail(email))
100         return {
101             error: true,
102             message: 'Mail inválido'
103         }
104
105     else if (!password || !validatePassword(password, 7, null, true, true)) {
106         return {
107             error: true,
108             message: 'Contraseña inválida'
109         }
110     }
111
112     return result;
113 }

```

Desafío de clase

Armar la función *validateRegisterUser* e implementarla. Testear su funcionamiento mediante el programa Bruno de solicitudes.

Lado cliente

Para el lado cliente, las validaciones nos van a servir tanto para el componente Login como para el componente Register, por lo cuál copiamos las *validations* de lado servidor y las pegamos en un archivo llamado *auth.services.js*, dentro de la carpeta *auth*.

Como ya poseíamos validaciones en React, las cambiaremos por las nuevas, además de modificar los *alert* por nuestros nuevos *errorToast*.

```

31  const handleLogin = (event) => {
32      event.preventDefault();
33
34      if (!emailRef.current.value.length || !validateEmail(email)) {
35          setErrors({ ...errors, email: true });
36          errorToast("¡Email incorrecto!");
37          emailRef.current.focus();
38          return;
39      }
40
41      else if (!password.length || !validatePassword(password, 7, null, true, true)) {
42          setErrors({ ...errors, password: true });
43          errorToast("¡Password incorrecto!");
44          passwordRef.current.focus();
45          return;
46      }
47
48      setErrors({ email: false, password: false })
49      onLogin();
50
51      fetch("http://localhost:3000/login", {
52          headers: {

```

Haremos lo mismo en la llamada de Register.

Mejoras

Manejo de errores

Actualmente, si el servidor nos devuelve un error, de todas maneras la ejecución del código cliente continua. Por ejemplo, en el Login, si el usuario es incorrecto igual lo envía a /library, solo que al no tener token asignado no puede acceder a ninguno de los recursos.

Idealmente, ante un error del servidor, mostraríamos ese error mismo al usuario, y cortaríamos el flujo de la aplicación. ¿Por qué no sucede esto, siendo que tenemos un catch escrito? Porque, justamente, al ser un error de servidor Javascript no lo registra y entiende que todavía la promesa se está resolviendo correctamente.

Para poder utilizar el *catch* de la fetch API, cambiaremos la *callback* intermedia (la que convierte en json la *response*) por:

```

51 fetch("http://localhost:3000/login", {
52   headers: {
53     "Content-type": "application/json"
54   },
55   method: "POST",
56   body: JSON.stringify({ email, password })
57 })
58 .then(async res => {
59   if (!res.ok) {
60     const errData = await res.json();
61     throw new Error(errData.message || "Algo ha salido mal");
62   }
63   return res.json();
64 }
65 )
66 .then(token => {
67   localStorage.setItem("book-champions-token", token)
68   navigate("/library");
69 }
70 )
71 .catch(err => {
72   errorToast(err.message)
73 }
74 )

```

Lo que hacemos es, primero confirmar que la respuesta es un error con `res.ok`, luego hacemos la conversión correspondiente y “tiramos” (comunicamos) que ha habido un error, con ese nuevo mensaje. Ese error que “tiramos”, si va a ser tomado en cuenta por Javascript y si va a salir por el `catch` (el cuál ahora lo encerramos en un `toast` para mejor UI).

Hacer lo mismo con todas las llamadas a servidor en nuestra app.

Nota: el uso de `await` / `async` en este caso surge como sugerencia de VS code, mas es posible también hacerlo con `res => {}`, es decir, una callback sin asincronía.

Separaciones en archivos adicionales

La siguiente mejora es sugerencia de la cátedra y no representa específicamente una buena práctica registrada de React.

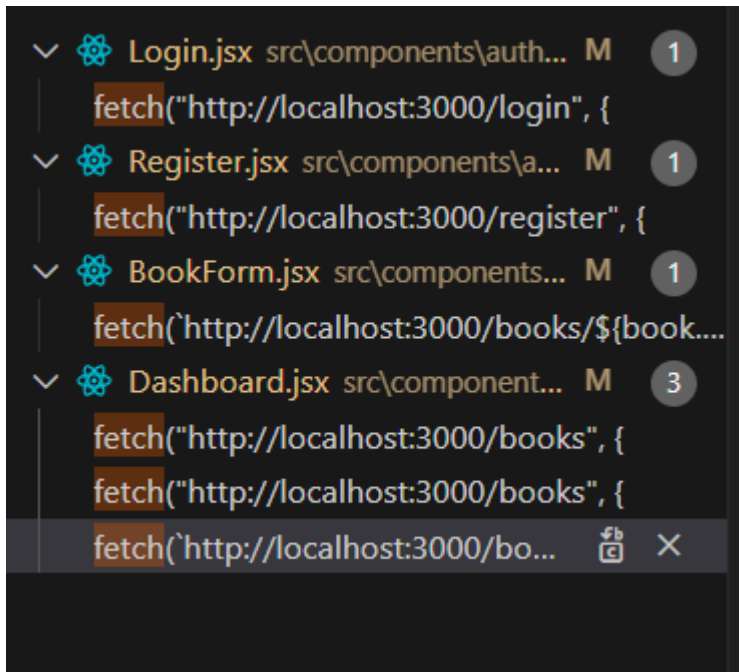
Los componentes de React solo tienden a crecer con el tiempo. Nuevas funcionalidades, partes nuevas de UI, etc. No solo crecen en el JSX sino que también en la parte de la lógica de eventos, en variables de ayuda, estilos. Nosotros proponemos una manera de dividir la carpeta del componente de la siguiente forma:

Supongamos que tenemos el componente `Dashboard.jsx` dentro de la carpeta `dashboard`. La cantidad de archivos (en el caso de que sea necesario) sería

- **Dashboard.jsx:** componente principal donde está todo el código referido a React (estado, `hooks`, manejadores de eventos, JSX)

- **Dashboard.css** o **Dashboard.scss**: los estilos para el componente.
- **Dashboard.helpers.js**: funciones de ayuda puramente de Javascript, que no utilizan código de React (pensemos por ejemplo, encontrar la media o la moda en un arreglo de números).
- **Dashboard.services.js**: llamadas al servidor.
- **Dashboard.data.js**: datos adicionales (por ejemplo, un estado inicial de un formulario)

Nos vamos a concentrar en la parte de services, que es de lo que más poseemos por fuera de React.



Seguimos el ejemplo con Dashboard. Movemos todas las funciones fetch al nuevo archivo Dashboard.services.js:

```

1  export const getBooks = (onSuccess, onError) => {
2    fetch("http://localhost:3000/books", {
3      headers: {
4        "Authorization": `Bearer ${localStorage.getItem("book-champions-token")}`
5      }
6    })
7    .then(async res => {
8      if (!res.ok) {
9        const errData = await res.json();
10       throw new Error(errData.message || "Algo ha salido mal");
11      }
12    })
13    .then(async res => {
14      return res.json();
15    })
16    .then(onSuccess)
17    .catch(onError);
18  }

```

```

19
20 export const addBook = (newBook, onSuccess, onError) => {
21   fetch("http://localhost:3000/books", {
22     headers: {
23       "Content-type": "application/json",
24       "Authorization": `Bearer ${localStorage.getItem("book-champions-token")}`
25     },
26     method: "POST",
27     body: JSON.stringify(newBook)
28   })
29   .then(async res => {
30     if (!res.ok) {
31       const errData = await res.json();
32       throw new Error(errData.message || "Algo ha salido mal");
33     }
34   })
35   .then(async res => {
36     return res.json();
37   })
38   .then(onSuccess)
39   .catch(onError);
40 }
41

```

```

43 export const deleteBook = (bookId, onSuccess, onError) => {
44   fetch(`http://localhost:3000/books/${bookId}`, {
45     method: "DELETE",
46     headers: {
47       "Authorization": `Bearer ${localStorage.getItem("book-champions-token")}`
48     }
49   })
50   .then(onSuccess)
51   .catch(onError);
52 }
53

```

Las funciones de *services* van a recibir los parámetros necesarios para que se ejecute la llamada al servidor, y luego dos parámetros adicionales: *onSuccess* y *onError*. Estas son

dos funciones *callbacks* que se van a ejecutar en caso de promesa resuelta o rechazada, respectivamente. Lo hacemos de esta manera ya que lo que sucede en el éxito o en el error **si corresponde a React, por ende tiene que estar dentro de un componente**. Ejemplo: los seteos de estados.

La llamada a *getBooks*, quedaría por ejemplo, de la siguiente manera:

```
17     useEffect(() => {
18         if (location.pathname === "/library") {
19             getBooks(
20                 data => setBookList([...data]),
21                 err => errorToast(err)
22             )
23         }
24     }, [location])
```

Y la llamada de agregar libros, así:

```
27     const handleBookAdded = (enteredBook) => {
28         if (!enteredBook.title || !enteredBook.author) {
29             errorToast('El autor y/o título son requeridos');
30             return;
31         }
32         addBook(
33             enteredBook,
34             data => {
35                 setBookList(prevBookList => [data, ...prevBookList]);
36                 successToast(`¡Libro ${data.title} agregado correctamente!`)
37             },
38             err => errorToast(err)
39         )
40     }
```

Realizar lo mismo para las llamadas de autenticación.

Variables de entorno

Las variables de entornos son variables que pertenecen, valga la redundancia, al entorno en el que se está desarrollando, accesibles mediante un archivo llamado *.env*. Recordemos que en general poseeremos varios entornos según la necesidad

- Local: entorno de desarrollo de nuestra PC.
- *Development*: entorno *live* donde podemos testear nuestra app los desarrolladores.
- *Staging*: entorno donde QA (el sector de testing) va a testear nuestra app.
- *Validation*: entorno donde generalmente el cliente o los PO (*product owners*) validaran la aplicación.

- *Production*: entorno para el público masivo.

En un archivo `.env.local` (con ese nombre, entra en el *gitignore*), vamos a guardar la variable `BASE_URL_SERVER` de nuestro servidor:

```
1 VITE_BASE_SERVER_URL=http://localhost:3000;
```

Las variables de entorno en vite deben tener el prefijo VITE para ser tomadas en cuenta.

Y luego la iremos cambiando en nuestros services:

```
1 const baseUrl = import.meta.env.VITE_BASE_SERVER_URL
2
3 export const getBooks = (onSuccess, onError) => {
4   fetch(`${baseUrl}/books`, {
5     headers: {
6       "Authorization": `Bearer ${localStorage.getItem("book-champions-token")}`
7     }
8   })
9   .then(async res => {
10     if (!res.ok) {
11       const errData = await res.json();
12       throw new Error(errData.message || "Algo ha salido mal");
13     }
14     return res.json();
15   })
16   .then(onSuccess)
17   .catch(onError);
18 }
19
20
21
```

Obviamente, según como salgamos a los distintos entornos con nuestra app (sea con Amazon Web Services, Google Cloud, Azure), definiremos variables de entorno.

Fecha	Versionado actual	Autor	Observaciones
16/02/2025	1.0.0	Gabriel Golzman	Primera versión