informe-login

Tomás Felipe Melli

Noviembre 2024

Índice

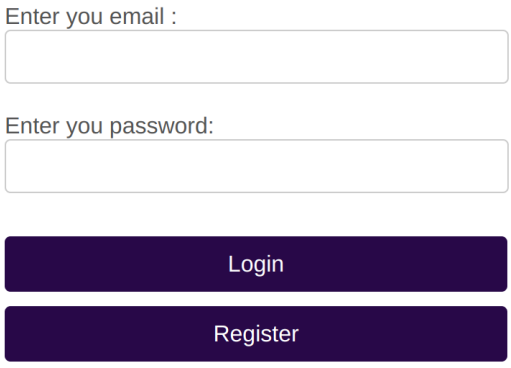
1 Introducción

En este informe se va a describir paso a paso como implementar el conjunto de estructuras y procesos que hacen posible una comunicación entre un usuario y una base de datos a través de la web. La idea, en resumidas cuentas es implementar una interfaz web en donde el usuario pueda registrarse o ingresar (dependiendo los permisos que tenga) al "sistema" que modelaremos. Para dicho modelo, necesitaremos constituir una base de datos con los campos necesarios : usuario -¿ contraseña. Además necesitamos definir el entorno del back-end, es decir, el espacio donde vamos a implementar la comunicación entre la anterior y el usuario (la API).
Las tecnologías a utilizar en este proyecto son las siguientes:

- **React.js**: una librería de JavaScript para el desarrollo de la interfaz de usuario.
- **Node.js**: entorno de ejecución para el back-end.
- **Express.js**: framework para crear la API.
- **MongoDB**: sistema de gestión de bases de datos NoSQL.
- **JWT (JSON Web Tokens)**: para los tokens de autenticación.
- **Postman**: para hacer las solicitudes HTTP y probar el buen funcionamiento de la API.
- **Axios**: una biblioteca de JavaScipt para hacer solicitudes HTTP desde el navegador.
- **Dotenv**: una herramienta de node para manejar variables de entorno.
- **CORS** (Cross-Origin Resource Sharing) para que no se restrinjan las solicitudes entre diferentes orígenes.

2 Front-End

La interfaz del usuario debe garantizar que el usuario pueda escribir sus datos (email y contraseña), ya sea para **registrarse** o **ingresar**.
La interfaz implementada es simple : dos espacios de *input :* $<$ *input* $><$ / $>$ uno con *type="text"* para la escritura del email y el otro con *type="password"* para ocultar la contraseña. Estos campos de input son obligatorios y están constrolados por dos botones. Como se ilustra en la siguiente imagen :



2.1 Definición de la clase e Inicialización

Para crear la Aplicación Web se hace uso de la clase App que hereda de Component funciones. Esto lo logramos en el momento de declarar el constructor de nuestra clase App en la primera parte : *super(props);* cuando llamamos al constructor de Component. Luego, hacemos uso del Objeto **state** que básicamente capturará las *propiedades* que tendrá nuestra componente. Inicializado, por supuesto en el constructor, le definimos los atributos : **email**, **password**, **error**,**mensaje**. Estos serán los valores que iremos capturando del usuario. La idea de este *state* es : frente a cualquier cambio, re-renderizar la interfaz.

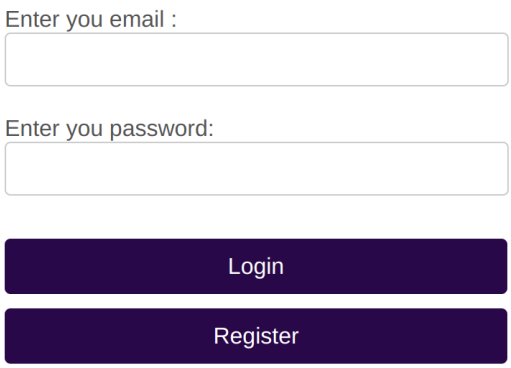
2.2 Métodos

Los métodos de nuestra clase serán :

- **cambios** es un método que como su nombre indica, será la responsable de capturar los cambios en los atributos : email y password de nuestra clase. Es decir, mantendrá referencia a los campos de *name="email"* y *name="password"* que ante cualquier cambio, *onChange={this.cambios}*, se ejecutará para actualizar los atributos de la clase. Estos cambios están representados por *value={this.state.email}*. Esta forma de declararlo nos da flexibilidad a la hora de manipular cambios tanto en el campo de email como en el de la password.
- **botón_login** es un método desarrollado exclusivamente para manejar la lógica del *login*. Como primera idea, el formulario no puede estar vacío, y por tanto, fallará en dicho caso, gracias al *preventDefault()*. Lo importante viene luego, con la implementación de un *try-catch* en el que intentaremos, asíncronamente, y gracias a *Axios*, realizar una solicitud HTTP de tipo POST a la ruta especificada */login* en la que enviamos el *email* y la *password* del state de la clase actual. Esta solicitud veremos en profundidad luego qué hace con esta información. Por ahora, vamos a considerar que la solicitud se responde, y esa respuesta la tenemos capturada. En caso de que el usuario con su respectiva constraseña exista en la base de datos, entonces se le enviará el *token* de seguridad y el *state* de la clase será modificado, es decir, **mensaje='Success'**. Caso contrario, se constata haber recibido una respuesta por parte de la solicitud. En caso de contener algo, se captura su *status* y *el mensaje de error* que serán utilizados para modificar el *state* de la clase de manera de mostrar el *error con el mensaje correspondiente y el estado de la petición*. Para el caso en que la respuesta sea NULL, se procederá a definir un mensaje por default en donde se modificará el *state* para que *error='Network error'*.
- **botón_registrarse** es un método que implementa la lógica del *registro*. Para ello, se realiza una solicitud de tipo POST a la ruta especificada */register*. Como ya mencionamos, se envía del *state*, los campos necesarios para el registro de un nuevo usuario. Asumimos que obtenemos una respuesta : en caso de éxito, el *state* se modifica de manera que *mensaje='Success'*; en caso de error, se modificará para que *error='error'*.
- **render** será la función de renderizado de la clase. Desglosemos sus componentes :
  - **Login** es el contenedor que engloba todo.
  - **Mensaje de éxito/error** usamos la idea de que : si el error no es NULL, entonces aparecerá en pantalla. ** Aclaración : en el momento de hacer un *setState* de la clase en las funciones relacionadas a las peticiones HTTP hay una lógica de *mensaje != NULL ¡==¿ error == NULL*. Aclarado esto, se entiende que sólo habrá mensaje si no hay error. Esto es lo que avisará al usuario si se pudo logear correctamente o no, y análogamente para la función de registro.
  - **Div de Email/Password** trivial.
  - **Botones de Registro/Login** también es trivial la explicación.

2.3 Estilos

Los estilos utilizados son meramente ilustrativos a fines del *Challenge*. Se implementó algo de movimiento en el botón como para hacer más suave la interfaz (*button:hover* y *cursor:pointer*) y en general el *borde redondeado* en los contenedores y botones. El uso de *verde* y *rojo* para denotar que hubo éxito/error en la petición también influye en el entendimiento de la interfaz.



2.4 Package.json

No es menor explicar las dependencias que se utilizaron; y por tanto, explicar que se agregó, como vimos, Axios y, dado que la estructura del proyecto fue creada vía *create-react-app* muchas de las dependecias fueron automáticamente añadidas. Se tuvo que modificar la sección del *"type"="module"* dado que Axios es compatible con *módulos ECMAScript* y por ello, la declaración por dafault imposibilitaba la correcta implementación del módulo. Por ello, veremos en el código que la forma de utilizar módulos será de la forma *import* {} *from " ";*.

3 Back-End

Por consigna, se nos pide diseñar la API que vincula todo lo comentado con una estructura de datos. Esta implementación será en vistas de un **Modelo Vista-Controlador** con la idea de modularizar las tareas para que sea más simple encontrar errores, en caso de tenerlos; o mismo, facilitar la escalabilidad del mismo. Dado que la consigna pide constatar credenciales de usuarios, retornar mensajes de éxito o de error según corresponda; tendremos por sección las siguientes implementaciones para dar solución a lo pedido :

3.1 Configuraciones

En la sección de **Configuraciones** se implementa toda la lógica de la base de datos. Para este challenge, se decidió usar **Mongo** pues, ya he trabajado en otra ocasión. La función **db_conexion** es la responsable de asíncronamente hacer la conexión. Podrá notarse lo siguiente en la implementación : **const conn = await mongoose.connect(process.env.MONGO_URI);** esta forma de realizar la conexión se debe a que es buena práctica almacenar datos sensibles en un **.env** como la *credencial de la base de datos* en este caso. Luego hablaré de qué otras cosas guardé allí cuando venga al caso.

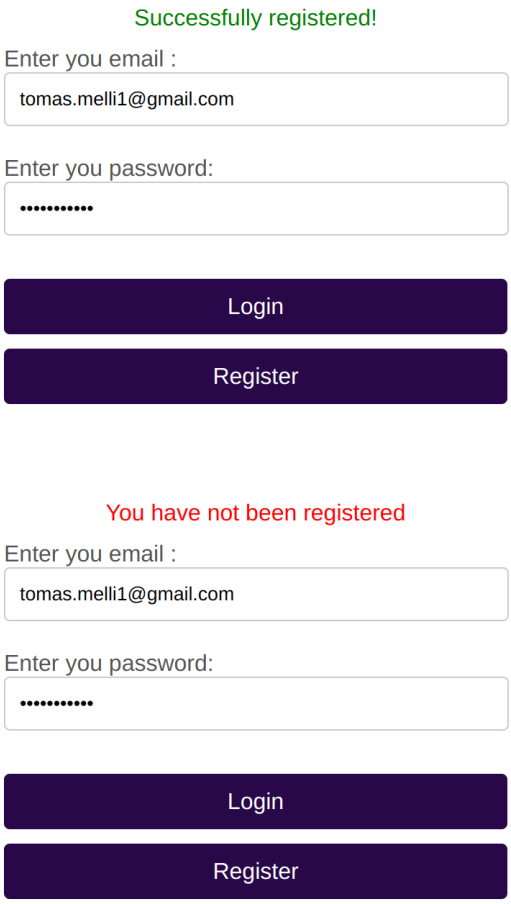
3.2 Modelo

En la sección del **Modelo** se implementó la lógica de la estructura del *usuario*, esto es, nuestro sujeto que hará uso del aplicativo. En este contexto es el **Schema**. El **UserScheme** que se usa es sencillo, un **email** que es de tipo string, que por *collection* en la DB debe ser *único (unique)* y para existir en la base de datos (creación o actualización) este campo es *obligatorio (required)*. Para la **password** pasa algo similar, tipo string y campo obligatorio.

3.3 Controladores

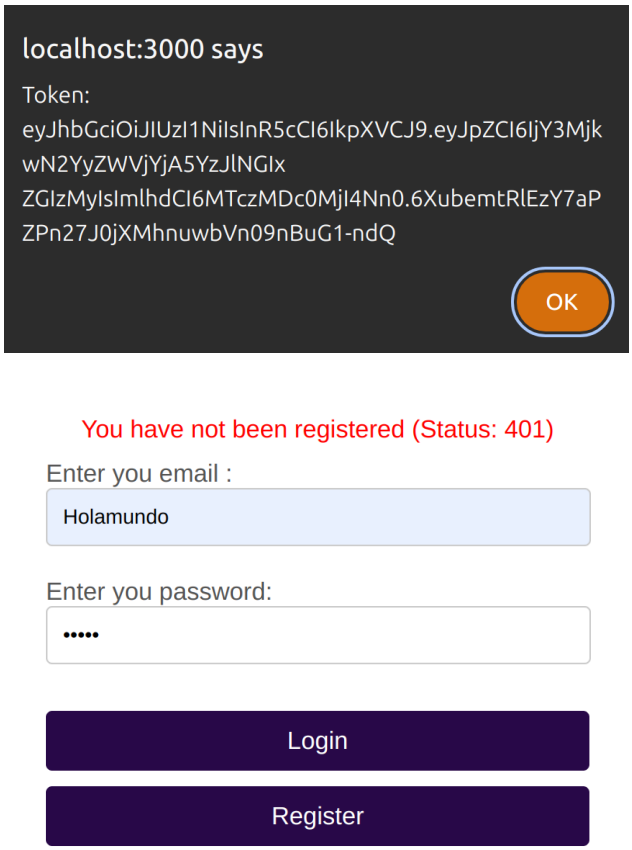
En la sección de controladores se implementa la lógica vinculada al manejo de las **peticiones**. Es decir, **qué pasa cuando se quiere registrar un usuario o logearse**. Veamos :

- **registrar_usuario** es la función encargada de controlar el registro del usuario. Es una función que recibe la *request* y *response* donde el primer parámetro es el *objeto* de la solicitud que viene del front (el email, password que mandamos ; y el segundo, el objeto *response* que tiene varios métodos asociados, a nosotros nos va a interesar el *status* para que reciba como respuesta eso el usaurio. Dicho esto, la lógica es : capturamos el *email* y la *password* del *body* de la request para poder crear el objeto *Usuario* definido previamente en el modelo. Procedemos a hacer el *save()* ya que se trata de una instacia de un *modelo de Mongoose*. Puede suceder que esto falle, y en dicho caso se captura el error y se manda como *status* (500 : estado de error del servidor) de la *response* y con un *send()* adjuntamos un mensaje también para que el usuario sepa. En el caso de éxito, se envía el estado (201 : estado HTTP creado) y también un pequeño mensaje.



- **login_user** es la función encargada de gestionar el login. Como mencionamos en la función anterior, se captura el *email* y la *password* y la lógica de esta función es corroborar si el usuario tiene credenciales para ingresar. Para ello, como *Usuario* es un modelo de Mongoose tiene el método *findOne(email)* que buscará en la collection el usuario asociado a ese mail (si existe). En caso de encontrarlo, verificará que coincida con el password enviado vía request.
  En caso de éxito, se geenra el **token** de seguridad con **jsonwebtoken**, utilicé este porque una amiga trabaja con este sistema y me recomendó. Este token es generado con la función *sign()* que básicamente toma dos parámetros : el **payload** que es el objeto con los datos de identificación del usuario que me gustaría almacenar para una potencial verificación de la identidad (en este caso tomé el _id de la base de datos que sabemos que es única y también el nombre del usuario, que también es única. Como

segundo parámetro, se le pasa la **clave secreta**, que como mencioné en la sección de configuración de la base de datos, la almacenaré de forma segura en *.env*. Esta se usa para firmar el token, es decir, para garantizar integridad.
Finalmente, en caso de éxito en la verificación de las credenciales de acceso, se devuelve el *status* (200: OK) y el *token*. En caso de *fallar en la autentcación*, se procede a devolver el estado (401 : No-Autorizado) con un pequeño mensaje (caso en que no esté registrado). Existe también el caso de *error* en donde algo sucede ajeno a la verificación, algo tal vez relacionado a la conexión, se devuelve (500 : error del servidor), pero para poder debbugear, en la consola de error nos guardamos el número de error.

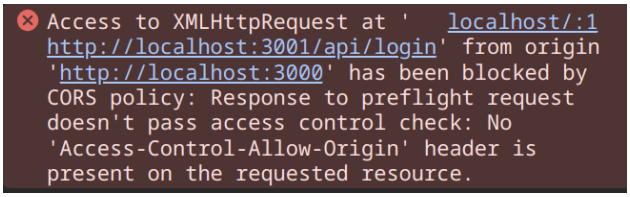


3.4 Rutas

En la sección de **Rutas** es donde se hace uso de**Express** para el **Routing**, es decir, manejar las solicitudes HTTP de manera que, dependiendo la ruta solicitada, se ejecute la función adecuada. En este caso, las solicitudes son de tipo **POST** (solicitudes de envío de información) ya que creamos usuarios o verificamos sus credenciales. Recordando la explicación de la función botón_login y botón_registrar, en la que definimos la ruta a la cuál se haría la solicitu, en este caso, definimos que cuando se haga la request, se responderá con una función u otra.

3.5 app.js

Finalmente, cómo confluye todo el sistema ? Quién ordena tanta modularización ? En **app.js** tenemos que concluir todo lo hecho. Se importan los módulos de *dotenv* para cargar las variables definidas en .env y se llama al método *config()* que las hace accesibles en *process.env* , *cors* (Cross-Origin Resource Sharing) para que no se restrinjan las solicitudes entre diferentes orígenes y que no suceda esto:



Importamos también la función de conexión a la base de datos de la sección *'./configuraciones'* así como las *rutas* definidas en la sección homónima. Se declara la variable *port* que hace referencia al puerto utilizado para que el servidor escuche. Y la parte fundamental : **la instancia de express** con el nombre *app*. Con esta instancia se hará todo. Es decir, se usa el método **use** para montar el **middleware** (es decir, cualquier función que intervenga durante una solicitud), por ejemplo, CORS, como vimos antes, todas las funciones que hablamos del login y register, necesitan ser montadas (la ruta es un parámetro opcional, que en el caso de las rutas, tenemos que especificar) y el uso de *express.json* es para facilitar el manejo de las solicitudes (como vimos que las capturábamos como *request.body*. lo hace más sencillo). Antes de terminar, se levanta el servido con el método **listen** en determinado puerto. Y en último lugar, se procede a establecer la conexión con la base de datos.

4 Conclusión

En conclusión, las consignas pedidas fueron cumplidas. Se desarrolló sobre cada componente utilizada, sobre la decisión de cada tecnología y se mostró la lógica de cada función. El código será anexado dentro del .zip del trabajo.

5 Anexo : uso de Postman

Se utilizó Postman para testear el correcto funcionamiento de la API.