«Talento Tech»

Node JS

Clase 15

Buenos Aires Ciudad

Clase N° 15 - Autenticación y Autorización

Temario:

1. Introducción a la Autenticación
2. JWT: Composición
3. Implementación de JWT en el Proyecto

Objetivos de la Clase

En esta clase, los estudiantes comprenderán la importancia de la autenticación en aplicaciones web explorando cómo los tokens JWT permiten la gestión segura de sesiones sin necesidad de almacenar información sensible en el servidor. Aprenderán a implementar middlewares para proteger rutas, asegurando que solo los usuarios autenticados accedan a ciertas partes de la aplicación, y analizarán buenas prácticas de seguridad para prevenir vulnerabilidades comunes, fortaleciendo así la integridad y confiabilidad de sus aplicaciones.

Introducción a la Autenticación

La autenticación es un pilar fundamental en el desarrollo de aplicaciones web, ya que permite verificar la identidad de los usuarios y garantizar que solo quienes tengan permisos adecuados puedan acceder a ciertos recursos.

Los ejemplos más comunes en el día a día los podemos observar en sistemas web que solicitan estar registrado mediante usuario y contraseña para poder acceder. Ya sea una aplicación de red social, tu app de música de preferencia o tu tienda favorita para comprar ropa, en todas posiblemente debas registrarte para obtener una experiencia completa y personalizada pero también para resguardar tu seguridad y evitar que otras personas puedan acceder a tus perfiles de usuario.

Cookies

En los inicios del desarrollo web, la autenticación se realizaba principalmente mediante el envío de credenciales en cada solicitud, lo que representaba un riesgo al exponer constantemente las contraseñas.

Para solucionar este problema, surgió el uso de cookies como mecanismo para almacenar información de autenticación en el navegador, permitiendo que los servidores identifiquen a los usuarios sin necesidad de reenviar credenciales en cada petición.

Las cookies son pequeños fragmentos de datos almacenados en el navegador del usuario y enviados con cada solicitud HTTP al servidor. Pueden contener información como identificadores de sesión o tokens de autenticación y se utilizan comúnmente para mantener el estado de autenticación en aplicaciones web. Sin embargo, su seguridad depende de cómo se configuren; las cookies pueden ser vulnerables a ataques como el secuestro de sesión si no se implementan medidas como la marca `HttpOnly`, `Secure` y el uso de `SameSite` para mitigar riesgos de ataques CSRF (cross site request forgery) o ataques de peticiones de origen cruzado.

Al ser una metodología insegura, con el paso del tiempo se fueron adoptando nuevas estrategias, dejando a las cookies como herramientas para almacenar preferencias del usuario como el modo de color o datos relacionados al marketing como último producto visitado.

El reemplazo de las cookies fue tomado en cierto modo por las sessions o sesiones de usuario que si bien también utilizan cookies, lo hacen de forma complementaria, resguardando la información de autenticación en el servidor.

Sessions

Las sesiones almacenan la información del usuario en el servidor y utilizan un identificador de sesión enviado al cliente mediante una cookie. Cada vez que el usuario realiza una solicitud, el servidor verifica el identificador y recupera los datos asociados a la sesión.

Este enfoque ofrece mayor seguridad en comparación con almacenar información sensible en el cliente, pero puede volverse costoso en términos de escalabilidad, ya que requiere almacenamiento y administración de sesiones activas, lo que puede ser un reto en aplicaciones con múltiples instancias del servidor.

Actualmente si bien es un enfoque un poco anticuado, aún se utiliza en la autenticación de algunos tipos de servicios y aplicaciones web.

Para lograrlo en un proyecto basado en Node JS y Express bastaría con instalar e implementar una librería de terceros para el manejo de sesiones de usuario como pueden ser `cookie-session` o `express-session`.

Third-party Authentication

Es la metodología de autenticación más extendida en el último. Esta se basa en la utilización de servicios de autenticación de terceros como pueden ser tu cuenta de **Google** o **Facebook** como credencial para acceder a una aplicación que no sea propiedad de estas empresas o contar con credenciales de tu propiedad pero que atraviesan un servicio de autenticación de terceros como puede ser **Okta** o **Auth0** que se encargan de todo el proceso de resguardo y validación.

Las ventajas de estos servicios radica en que la responsabilidad por la integridad de la información y la seguridad del sistema de autenticación implementado corren por cuenta de estos y te olvidas de contar con una capa específica en tu aplicación con este fin.

En muchos casos puede resultar una alternativa ideal para enfocar tu desarrollo en las funcionalidades principales de tu aplicación sin preocuparte demasiado por la capa de autenticación aunque también, dependiendo tus necesidades de negocio, puedas necesitar tener un sistema robusto de autenticación hecho a la medida.

JSON Web Tokens (JWT)

JSON Web Token (JWT) es un estándar abierto (**RFC 7519**) que define una forma compacta y autónoma de transmitir información de manera segura entre las partes como un objeto JSON. Esta información puede ser verificada y confiable porque está firmada digitalmente. Los JSON Web Tokens representan una solución más moderna a las sesiones, basada en un enfoque sin estado.

Un **JWT** es un token en formato **JSON** que contiene información codificada y firmada digitalmente, lo que permite que el servidor valide su autenticidad sin necesidad de consultar una base de datos o mantener una sesión activa.

Al ser transportado en los encabezados de las solicitudes HTTP, permite que las aplicaciones sean más escalables y fáciles de integrar con APIs y sistemas distribuidos. No obstante, su implementación requiere precaución, como establecer tiempos de expiración adecuados y evitar almacenar información sensible en el payload, ya que los JWT pueden ser interceptados si no se protegen adecuadamente.

Esta metodología de autenticación es ideal para servicios del tipo API Rest, ya que permiten enviar **tokens** mediante los **headers** o cabeceras de las peticiones que serán capturadas por los middlewares al llegar a las rutas y validarán si el acceso a los recursos debe ser permitido.

¿Cómo funciona JWT?

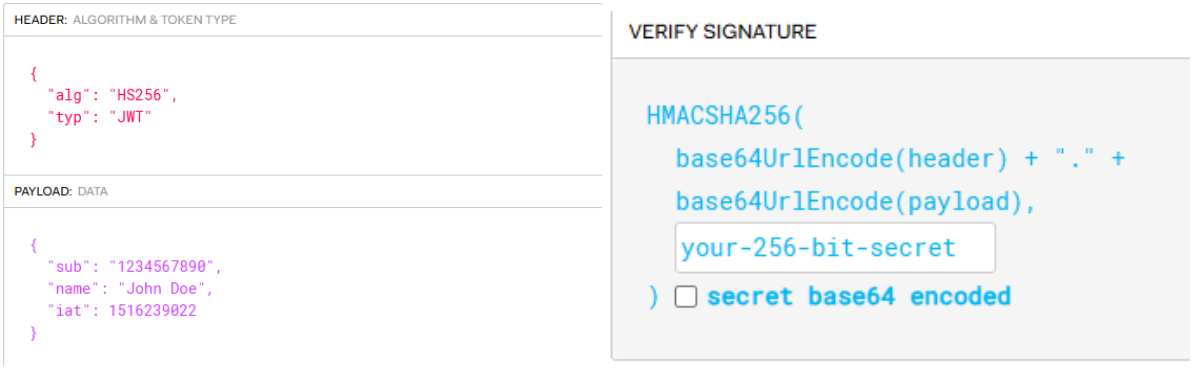
- Generación del Token: Cuando un usuario inicia sesión con sus credenciales, el servidor genera un JWT y lo envía al cliente.
- Almacenamiento del Token: El cliente almacena el token (por ejemplo, en localStorage o cookies).
- Envío del Token: En cada solicitud subsecuente, el cliente envía el token en el encabezado de autorización.
- Verificación del Token: El servidor verifica el token y, si es válido, permite el acceso a los recursos protegidos.

JWT: Composición

Un **JWT** se compone de tres partes codificadas en Base64: el encabezado, que especifica el algoritmo de cifrado y el tipo de token; el payload, que contiene los datos del usuario y las reclamaciones o "claims"; y la firma, que asegura la integridad del token y evita modificaciones malintencionadas.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

*Versión **codificada** de un JWT*



*Versión **decodificada** de un JWT*

Su uso es sencillo: cuando un usuario inicia sesión correctamente, el servidor genera un JWT y lo envía al cliente, quien lo almacena (normalmente en el almacenamiento local o en una cookie segura). En cada solicitud a rutas protegidas, el cliente envía el token en el encabezado de autorización, permitiendo al servidor verificar la validez del mismo y conceder o denegar el acceso según corresponda.

En el caso de nuestra API, no contamos con una vista de login y normalmente este tipo de llaves de acceso se solicitan creando una cuenta en dicho servicio para poder consumir esa información. Sitios como Mercadolibre o la NASA cuentan con APIs públicas pero para poder utilizarlas se necesita crear una cuenta y pedir un Token para comenzar a realizar peticiones.

En nuestro caso, crearemos una capa de “autenticación” con una ruta de `/login` que validará las credenciales de un usuario ficticio.

Implementación de JWT en el Proyecto

Instalación de dependencias y configuración

El primer paso es instalar las dependencias `jsonwebtoken` y `body-parser`:

```
npm install jsonwebtoken body-parser
```

Además utilizaremos nuestro archivo `.env` donde guardaremos una llave secreta para firmar nuestros TOKENS:

```
JWT_SECRET_KEY=jwt_secret_key
```

Esta llave secreta puede ser cualquier valor que consideremos apropiado como por ejemplo: `api_rest_project_secret_token_key`.

Crear un nuevo TOKEN

Dentro del la ruta `src/utils` creamos un archivo llamado `token-generator.js` donde crearemos la función para la generación de nuevos tokens:

```
import jwt from 'jsonwebtoken';
import 'dotenv/config';

const secret_key = process.env.JWT_SECRET_KEY;

// Función para generar un token JWT
export const generateToken = (userData) => {

  const user = {id: userData.id, email: userData.email};
  const expiration = { expiresIn: '1h' };

  return jwt.sign(user, secret_key, expiration);
}
```

En el ejemplo anterior creamos la función llamada `generateToken` el cual recibe la información del usuario y donde definimos mediante la propiedad `expiresIn` el tiempo de vida del `token` creado. Luego utilizamos la librería de `jwt` donde mediante su método

`sign`, y las variables de `user`, `secret_key` y `expiration` generamos y retornamos un nuevo `tokenID`.

Una vez que tenemos una función para crear `tokens`, vamos a necesitar una capa funcionalidad nueva de nuestra aplicación que permita identificar usuarios y crear tokens para que puedan acceder a los recursos de nuestra API.

Para ello crearemos un nuevo router en la ruta `src/routes/auth.routes.js` donde atenderemos al endpoint `/login` mediante el método `POST`:

```
// auth.routes.js

import express from 'express';
import { login } from '../controllers/auth.controller.js';

const router = express.Router();

router.post('/login', login);

export default router;
```

También necesitaremos crear el controlador en `src/controllers/auth.controller.js`:

```
  import { generateToken } from '../utils/token-generator.js';

  const default_user = {
    id: 1,
    email: "user@email.com",
    password: "strongPass123"
  }

  export async function login(req, res) {
      const { email, password } = req.body;

      // Aquí deberías verificar las credenciales del usuario

      // Ejemplo de usuario autenticado
      const user = { id: 1, email };

      if (email === default_user.email
         && password === default_user.password) {

          const token = generateToken(user);
```

```
            res.json({ token });
        } else {
            res.sendStatus(401);
        }
}
```

Como podemos observar, contamos con un objeto que contiene la simulación de la consulta de los datos de un usuario real a la base de datos que son los que usaremos para comparar con las credenciales recibidas en la petición.

De esta manera, el cliente realizará una petición a nuestra ruta de `/login` enviando en el body de la misma el `email` y la `password` que deberán coincidir con un usuario real registrado, en este caso, nuestro `default_user`.

Una vez obtenemos los datos mencionados desde el `body` de la petición, comparamos con "la consulta realizada en nuestra base de datos" (`default_user`) y en caso de coincidir las credenciales, creamos un token para este usuario mediante `generateToken`. Luego lo devolvemos en la response para que a partir de ahora el cliente pueda realizar peticiones a nuestra API con un token válido, obteniendo acceso a los recursos protegidos.

En caso que las credenciales no sean válidas, devolvemos el `status 401 "Unauthorized"` y podríamos agregar un mensaje en formato JSON explicando el error.

Finalmente, resta agregar nuestro router al archivo entrypoint `index.js`:

```
import authRouter from './src/routes/auth.routes.js';

app.use(bodyParser.json());

// Routers
app.use('/auth', authRouter);
```

También agregamos el middleware global de `bodyParser.json()` esto nos va a permitir leer el `body` de las peticiones cuando lo recibimos en formato JSON, caso contrario nuestro programa no lo entendería.

Middleware de autenticación

Ahora nos toca crear el `middleware` encargado de validar el `token` enviado por los clientes ya autenticados. En la ruta `src/middlewares` creamos un archivo llamado `authentication.js` y le damos forma a nuestro middleware:

```
import jwt from 'jsonwebtoken';
import 'dotenv/config';

const secret_key = process.env.JWT_SECRET_KEY;

// Middleware para verificar el token JWT
export const authentication = (req, res, next) => {
  const token = req.headers['authorization'].split(" ")[1];

  if (!token) return res.sendStatus(401);

  jwt.verify(token, secret_key, (err) => {
      if (err) return res.sendStatus(403);
      next();
  });
}
```

En primer lugar leemos el `token` recibido en el header de `authorization` que tendrá un formato parecido a `“Bearer iJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6…”` por lo cual es importante separar la cadena de texto mediante el “ “ y tomar el valor en la posición `[1]` donde se encuentra el código en base64.

En caso que el token no exista o haya venido en formato inválido devolveremos un `status 401 “Unauthorized”`. Por otro lado, si el formato es correcto procedemos con la verificación del `token` mediante `jwt.verify` donde si la verificación arroja algún error significa que el `token` no es válido y retornamos un `status 403 “Forbidden”`, caso contrario devolvemos `next()` certificando la validez del `token` y continuando con la ejecución del programa.

Protección de rutas

Una vez que tenemos nuestro middleware es momento de proteger nuestras rutas. Para esto existen distintas maneras:

- Podemos utilizar nuestro middleware en el archivo entrypoint `index.js` como un middleware global:

  ```
  app.use(authentication);
  ```

  De esta manera estaríamos protegiendo todas las rutas de nuestra aplicación, lo cual en este caso no es lo recomendable ya que necesitaremos una ruta sin protección para generar los `tokens` por parte de los usuarios.

- También es posible proteger por conjuntos de rutas, en el mismo archivo pero en lugar de hacerlo global, seleccionamos los conjuntos a autenticar:

  ```
  app.use('/auth', authRouter)
  app.use('/api', authentication, productsRouter);
  app.use('/categories', authentication, categoriesRouter);
  ```

  Nótese que en el ejemplo elegimos proteger las rutas que permiten acceder a los endpoints de productos y de categorías, pero no a las rutas de `/auth`.

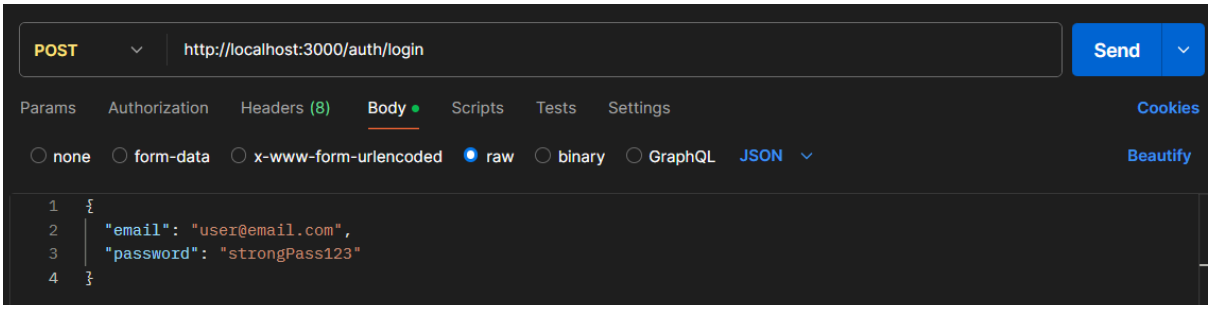
- La última es proteger rutas de forma individual dentro de los archivos de cada router:

  ```
  router.get('/products', authentication, getAllProducts);
  router.get('/products/:id', authentication, getProductById);
  router.post('/products', authentication, createProduct);
  ```

Pruebas con POSTMAN

Listo, ya tenemos configurada la capa de autenticación y autorización de nuestra API Rest, ahora nos queda validar que todo funcione correctamente y para ello utilizaremos POSTMAN.

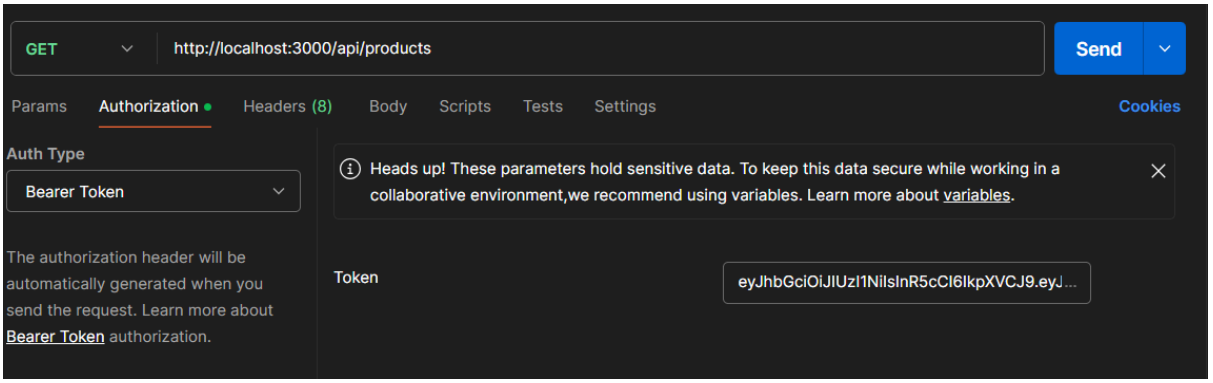
El primer paso el crear un token válido ingresando nuestras credenciales de cuenta:



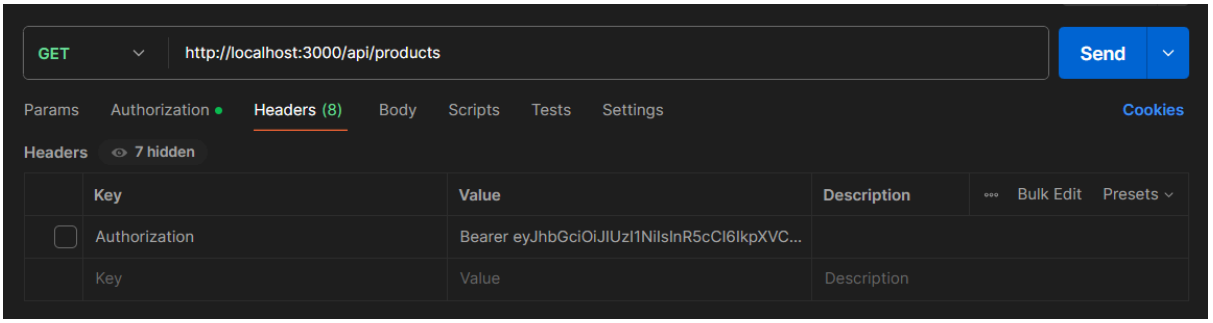
Creamos una petición mediante el método `POST`, y apuntamos a la ruta `/auth/login` agregando en el `body` las credenciales de `default_user` en formato JSON.

Una vez presionemos **send**, en caso de enviar las credenciales correctas, obtendremos nuestro token, el cual usaremos para nuestras peticiones:

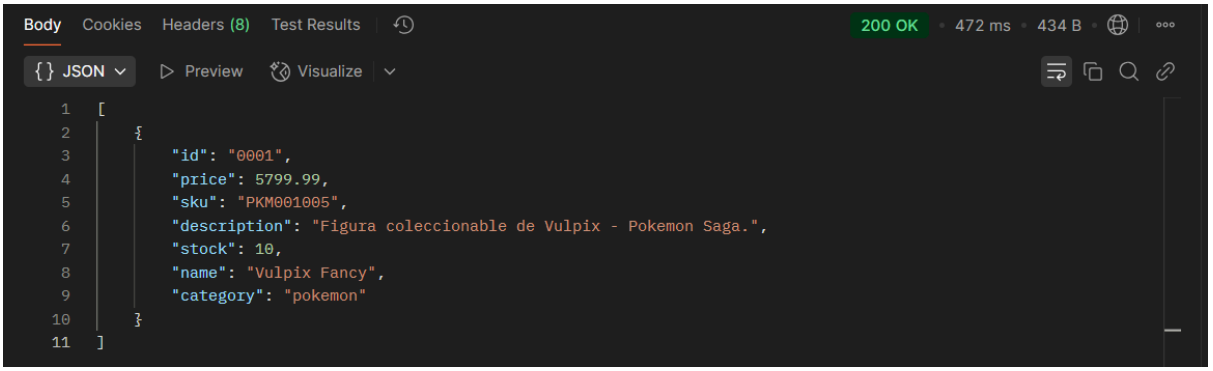
Ahora que ya obtuvimos un `token` válido, es momento de ver si tenemos acceso a los productos de la API Rest. Para eso realizaremos una petición mediante el método `GET` a la ruta `api/products` y en la pestaña de "**Authorization**" seleccionamos la opción "**Bearer Token**" y colocamos el `token` del paso anterior:



Otra forma, es desde la pestaña “**Headers**” colocar una nueva cabecera llamada `Authorization` con el valor “**Bearer** ” seguido de nuestro `token`**:**



Una vez realizada la consulta y si hemos seguido los pasos anteriores correctamente obtendremos los datos solicitados:



Del mismo modo podemos probar con todas las demás rutas protegidas que hayamos configurado en nuestra API Rest.

Ejercicio Práctico

Autenticación con JWT: Protegiendo la API

Después de completar la integración con Firestore, Sabrina y Matías regresan con un nuevo desafío.

"La API ahora maneja datos en la nube, pero ¿qué pasa si cualquier persona intenta acceder a ellos? Es momento de dar el siguiente paso y proteger nuestra aplicación."

"En aplicaciones reales, los datos deben estar resguardados. Para ello, utilizaremos JSON Web Tokens (JWT), un estándar seguro para manejar autenticación en APIs."

Misión:

1. **Instalar y configurar JWT:**
    - Agrega las dependencias **jsonwebtoken** y **body-parser** a tu proyecto.
    - Crea las variables de entorno necesarias para almacenar tu clave secreta.
    - Implementa la lógica para generar un token JWT cuando un usuario inicie sesión correctamente.

2. **Middleware de acceso:**
    - Crea una función middleware que intercepte las peticiones antes de que lleguen a los controladores.
    - Extrae el token desde el header Authorization y verifica su validez utilizando la lógica de JWT.
    - Si el token es válido, permite el acceso; de lo contrario, devuelve un error de autenticación.

3. **Implementar la Validación de Usuarios**
    - Aplica el middleware en las rutas que requieran autenticación (por ejemplo, rutas para obtener o modificar datos en Firestore).
    - Prueba el flujo enviando peticiones con y sin tokens válidos para asegurarte de que el sistema funcione correctamente.

"Cuando completes este reto, nuestra API será mucho más segura y profesional. ¡Ahora estás protegiendo tu aplicación como un verdadero desarrollador backend!"

Consignas de Proyecto Final

La fecha de lanzamiento de nuestra aplicación está cada vez más cerca. Sabrina y Matías, emocionados por ver el resultado de tu esfuerzo, quieren recordarte los aspectos clave que el sistema debe cumplir antes de su despliegue en producción.

Matías toma la palabra con una sonrisa:

“Este es tu primer proyecto como parte de TechLab, y estamos realmente impresionados con tu desempeño. Ahora nos queda el paso más crucial: la salida a producción.”

Sabrina asiente y añade con confianza:

“Antes de avanzar, debemos asegurarnos de que nuestra aplicación cumpla con todos los requisitos establecidos. Confiamos en tu trabajo y sabemos que todo estará listo para el gran lanzamiento.”

Premisa

Actualmente nuestro cliente tiene diversos productos en catálogo y precisa disponer de una API Rest desde donde su tienda oficial pueda administrarlos, habilitando la posibilidad de Leer, Crear, Actualizar y Eliminar la información sobre los productos.

La aplicación debe contar con una capa de autenticación para resguardad la seguridad de los datos que estarán alojados en una base de datos en la nube mediante el servicio Firestore de Firebase.

Es importante definir una arquitectura escalable, separando las distintas responsabilidades de la aplicación en capas que permitan establecer rutas, controladores, servicios y modelos de forma clara y prolija, además de las carpetas necesarias para guardar middlewares y configuración a servicios externos.

Finalmente, la aplicación debe contemplar el manejo de errores de forma clara, teniendo en cuenta fallos del tipo 404 para rutas no definidas, los estados 401 y 403 ante errores de autenticación y códigos de estado 400 y 500 cuando las peticiones contienen errores o nuestros servicios externos de datos no responden.

Requerimientos del Proyecto

Requerimiento #1: Configuración Inicial

- Crea un directorio donde alojarás tu proyecto e incluye un archivo `index.js` como punto de entrada.
- Inicia Node.js y configura npm usando el comando `npm init -y`.
- Agrega la propiedad `"type": "module"` en el archivo `package.json` para habilitar ESModules.

Configura un script llamado `start` para ejecutar el programa con el comando `npm run start`.

Requerimiento #2: Instalación de dependencias

- Instala `express`, `cors`, `body-parser`, `dotenv`, `firebase` y `jsonwebtoken` como dependencias del proyecto.

Requerimiento #3: Configuración del servidor

- Crea un servidor web con `express` y realiza su configuración en el archivo `index.js`.
- Configura CORS para habilitar las peticiones de origen cruzado, así las aplicaciones Frontend de la empresa pueden consultar al servicio sin problemas.
- Configura el middleware global de body-parser para interpretar los body en formato JSON de las peticiones.
- Establece un middleware que maneje las rutas desconocidas, devolviendo el estado 404 y un mensaje.
- Crea un archivo `.env` donde se alojarán las variables de entorno del proyecto.

Requerimiento #4: Rutas

- Crea la capa de rutas del proyecto.
- Establece las rutas necesarias para atender las peticiones que interactúan con productos, así como también la ruta de login para autenticar usuarios:
    - `products.routes.js`:

      `GET /api/products` devuelve todos los productos.
      `GET /api/products/:id` devuelve el producto con el ID indicado.
      `POST /api/products/create` recibe en el cuerpo (body) de la petición la información sobre el nuevo producto para ser guardado en el servicio de datos en la nube.
      `DELETE /api/products/:id` elimina el producto con el ID indicado.

    - `auth.routes.js`:

`POST /auth/login` recibe las credenciales de usuario en el cuerpo (body) de la petición y devuelve el Bearer token si son válidas o un error de autenticación en caso contrario.

Requerimiento #5: Controladores y Servicios

- Crea la capa de controladores para cada una de las rutas establecidas en el requerimiento anterior.
- Crea la capa de servicios para atender a cada uno de los controladores.

Requerimiento #6: Acceso a los datos

- Crea la capa de modelos de la aplicación.
- Crea un nuevo proyecto de Firestore en Firebase, agrega una colección para registrar nuevos productos y crea el primer documento de producto para darle estructura y tipos de datos.
- Configura y conecta Firebase en el proyecto.
- Utiliza la instancia de Firebase creada y crea los métodos necesarios para que el modelo pueda interactuar con la base de datos remota.
- Conecta los servicios con los modelos.

Requerimiento #7: Protege tus rutas

- Configura JWT en el proyecto
- Crea un middleware de autenticación y protege las rutas correspondientes
- Agrega la lógica necesaria en la en controlador de login para validar la identidad del usuario y devolver un Bearer Token.

Requerimiento #8: Despliegue a producción

- Configura tu archivo vercel.json
- Crea un nuevo proyecto en vercel
- Despliega tu proyecto a producción

"El éxito de este proyecto está en tus manos. Es momento de dar el último paso y asegurarnos de que todo esté listo para el gran lanzamiento. ¡Confíamos en vos!"

Materiales y Recursos Adicionales:

Documentación Oficial de JSON Web Tokens (JWT):
jwt.io – Explicación del estándar, estructura del token y herramientas para decodificarlo.

Uso de JSON Web Tokens en Node.js:
Guía oficial de jsonwebtoken – Documentación de la librería utilizada para firmar y verificar tokens en Node.js.

Codificación Base64 en MDN – Explicación de cómo funciona Base64 y su uso en la web.

Preguntas para Reflexionar:

- ¿Por qué es importante almacenar de forma segura la clave secreta utilizada para firmar los tokens?
- ¿Cómo garantizas que los tokens generados sean seguros y no puedan ser alterados por terceros?
- ¿Cuáles son los riesgos de utilizar JWT sin configurar un tiempo de expiración adecuado?
- ¿Qué rutas de tu API deberían estar protegidas y cuáles pueden mantenerse públicas?
- ¿Qué problemas encontraste al validar tokens en tu API y cómo los solucionaste?

Próximos Pasos:

- **Fin de la cursada**: daremos cierre al curso mediante la presentación de proyectos finales.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad