

Guía de estudio 6 - Autenticación y Autorización de usuarios con JWT



¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo practicar y ejercitar los contenidos que hemos visto en clase, además de profundizar temas adicionales que complementan aquellos vistos en la clase.

¡Vamos con todo!



Tabla de contenidos

Autenticación y Autorización	3
Autorización con JWT	5
Setup del proyecto	5
Generación de tokens JWT	6
¡Manos a la obra! - Token con JWT	9
Obteniendo un token en las cabeceras	10
¡Manos a la obra! - PUT /eventos/:id	11
Verificación y Decodificación de un JWT	11
¡Manos a la obra! - Función actualizarEvento	13
Encriptado de contraseñas	14
Registro de usuarios	14
Inicio de sesión de usuarios	16
¡Manos a la obra! - Nuevo usuario	17
Límite de tiempo en token	17
¡Manos a la obra! - Nombre de la actividad	19
Preguntas:	19



¡Comencemos!

Autenticación y Autorización

Implementar sistemas de usuarios en nuestras aplicaciones conlleva a entender 2 conceptos parecidos:

La **autenticación** consiste en Identificar la existencia de usuarios según las credenciales recibidas.

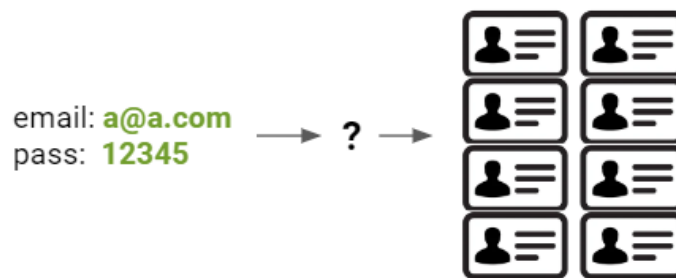


Imagen 1. Autenticación de usuarios
Fuente: Desafío Latam

Mientras que la **autorización** consiste en autorizar usuarios para acceder a información o funcionalidades restringidas



Imagen 2. Autorización de usuarios
Fuente: Desafío Latam

Un formulario de login inicia el proceso de una autenticación en un sistema

Formulario de login de la plataforma {desafío} latam_. El formulario está sobre un fondo oscuro y contiene el logo a la izquierda. A la derecha, hay dos campos de entrada: 'Usuario' y 'Contraseña'. El campo de contraseña tiene un icono de ojo para alternar la visibilidad. Debajo de los campos, hay dos botones: 'Recuperar cuenta' y 'Ingresar'. En la parte inferior izquierda del formulario, se lee: 'Comienza el desafío de {desarrollar tu futuro}'.

Imagen 3. Formulario de Login de ejemplo sobre la Autenticación
Fuente: Desafío Latam

Al presionar el botón ingresar, se procede a verificar que las credenciales escritas efectivamente existen en la base de datos y posterior a esto, se le entrega la autorización al cliente para acceder a secciones restringidas de la página web

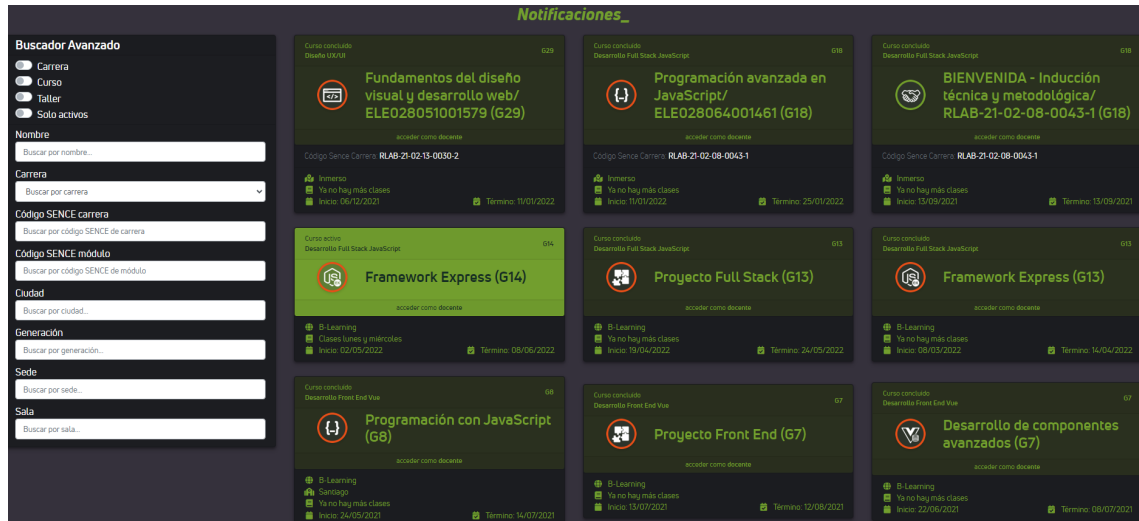


Imagen 4. Acceso a un sistema luego de una autorización
Fuente: Desafío Latam

El orden de operaciones entonces sería el siguiente:



Imagen 5. Orden de operaciones Visita - Autenticación - Autorización - Acceso
Fuente: Desafío Latam

Una vez que el usuario obtiene la autorización del servidor, podrá acceder a información y funcionalidades restringidas sin necesidad de volver a autenticarse.

El usuario solo deberá entregar la llave recibida para acceder a donde requiera.

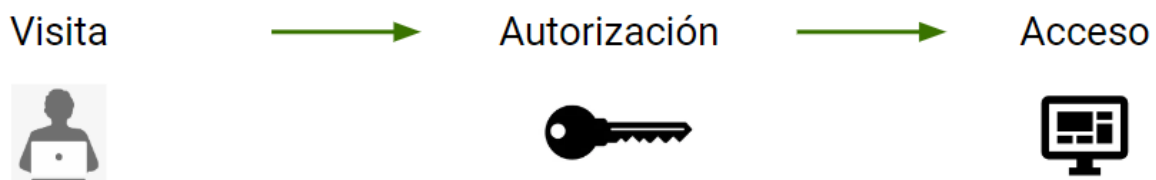


Imagen 6. Acceso luego de una autenticación y autorización previa
Fuente: Desafío Latam

Autorización con JWT

Cuando un usuario ingresa con sus credenciales válidas recibirá un token, luego solo tiene que mostrar este para saber si tiene acceso o no.

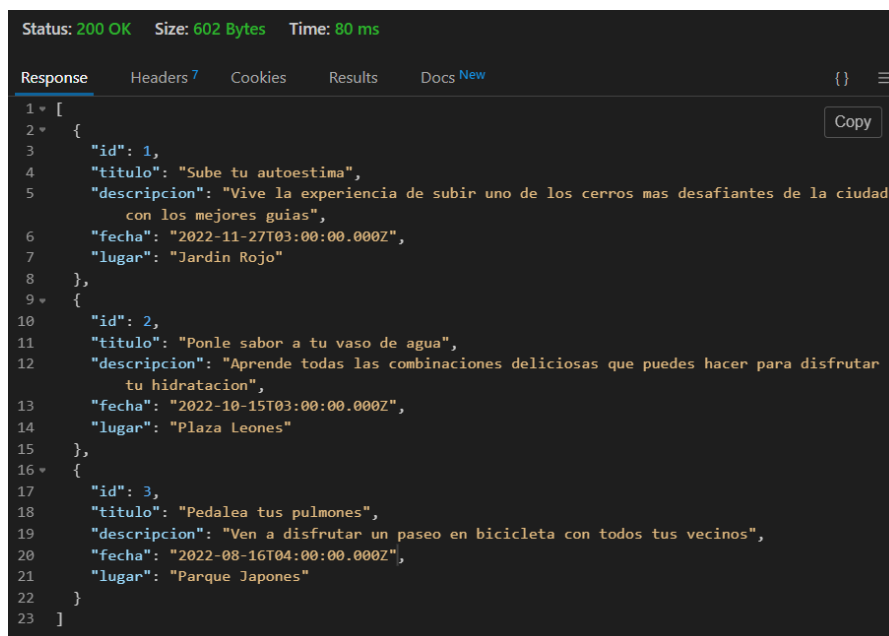
Para generar los tokens utilizaremos JSON Web Token, abreviado **JWT** que nos provee de métodos para la generación, validación y decodificación de tokens.

En esta unidad vamos a crear el backend de un sistema que autentica y autoriza usuarios.

Setup del proyecto

Para enfocarnos exclusivamente en la autenticación y autorización descargaremos un proyecto base llamado Vida Sana.

Una vez descargado, ejecuta el script SQL para crear la base de datos que utilizaremos, luego levanta el proyecto y prueba la ruta GET /eventos con ThunderClient u otro cliente para probar APIs o también puedes ocupar directamente el navegador. Confirma que se obtienen con éxito los registros de la base de datos.



```
Status: 200 OK Size: 602 Bytes Time: 80 ms
Response Headers 7 Cookies Results Docs New {} ≡
1 * [
2 * {
3   "id": 1,
4   "titulo": "Sube tu autoestima",
5   "descripcion": "Vive la experiencia de subir uno de los cerros mas desafiantes de la ciudad con los mejores guias",
6   "fecha": "2022-11-27T03:00:00.000Z",
7   "lugar": "Jardin Rojo"
8 },
9 * {
10  "id": 2,
11  "titulo": "Ponle sabor a tu vaso de agua",
12  "descripcion": "Aprende todas las combinaciones deliciosas que puedes hacer para disfrutar tu hidratacion",
13  "fecha": "2022-10-15T03:00:00.000Z",
14  "lugar": "Plaza Leones"
15 },
16 * {
17  "id": 3,
18  "titulo": "Pedalea tus pulmones",
19  "descripcion": "Ven a disfrutar un paseo en bicicleta con todos tus vecinos",
20  "fecha": "2022-08-16T04:00:00.000Z",
21  "lugar": "Parque Japones"
22 }
23 ]
```

Imagen 7. Eventos registrados en la base de datos (Recurso público)
Fuente: Desafío Latam

La **temática** del proyecto que descargaste es la de una empresa de eventos llamada Vida Sana, que constantemente invita a la comunidad a participar en sus actividades para una vida más sana.

Los eventos serán **recursos públicos** que cualquier cliente puede consumir, mientras que las modificaciones y eliminaciones de los eventos deberán ser una acción **solo disponible para usuarios autorizados** con JWT

Utilizaremos el archivo index.js para la creación de nuevas rutas, firma, verificación y decodificación de tokens.

Mientras que en consultas.js escribiremos todo lo relacionado con las funciones que interactúan con la base de datos

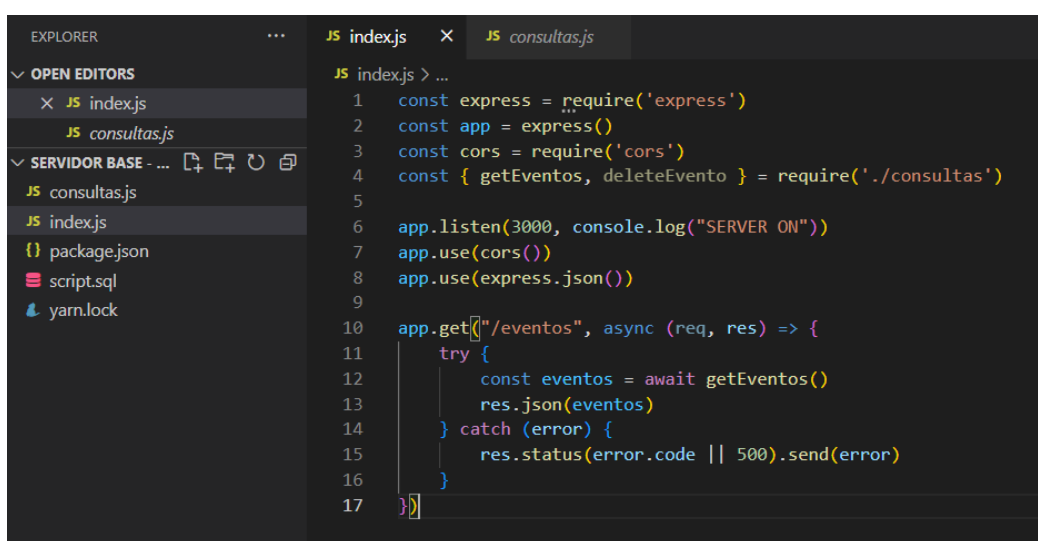


Imagen 8. Servidor Base disponible en la plataforma
Fuente: Desafío Latam

Generación de tokens JWT

Nuestro primer paso será generar tokens, para esto entramos al proyecto descargado y con npm instalaremos la biblioteca **jsonwebtoken**

```
npm install jsonwebtoken
```

Luego, debemos importar el paquete al comienzo del código del servidor base, en el index.js:

```
// index.js
const jwt = require("jsonwebtoken")
```

Nuestro siguiente paso será crear una función que realice el proceso de autenticación de un usuario en `consultas.js`

En caso de que las credenciales recibidas no coincidan con ningún registro de la tabla, deberemos devolver un error detallando lo sucedido.

```
const verificarCredenciales = async (email, password) => {
  const consulta = "SELECT * FROM usuarios WHERE email = $1 AND password = $2"
  const values = [email, password]
  const { rowCount } = await pool.query(consulta, values)
  if (!rowCount)
    throw { code: 404, message: "No se encontró ningún usuario con estas credenciales" }
}
```

Ahora agreguemos ruta **POST /login** que reciba las credenciales de un usuario y utilice la función **verificarCredenciales** para confirmar su existencia y posteriormente generar y devolver un token **JWT**

```
app.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body
    await verificarCredenciales(email, password)
    const token = jwt.sign({ email }, "az_AZ")
    res.send(token)
  } catch (error) {
    console.log(error)
    res.status(error.code || 500).send(error)
  }
})
```

La generación de un token consiste en ejecutar el método **sign** de la instancia **jwt**

```
const token = jwt.sign(<payload>, <llave secreta>)
```

El primer argumento de este método es el **payload** que existirá dentro del token cifrado.

El segundo argumento es la **llave secreta**, la cual debe ser usada para decodificar posteriormente este u otro token generado con la misma llave.



La **llave secreta** es un dato de estricta importancia que debe ser guardado con mucho cuidado, quien la tenga podrá firmar nuevos tokens accediendo a recursos y funcionalidades restringidas.

Probemos nuestra nueva ruta realizando una consulta con Thunder Client pasando como payload las credenciales de uno de los usuarios registrados en la base de datos.

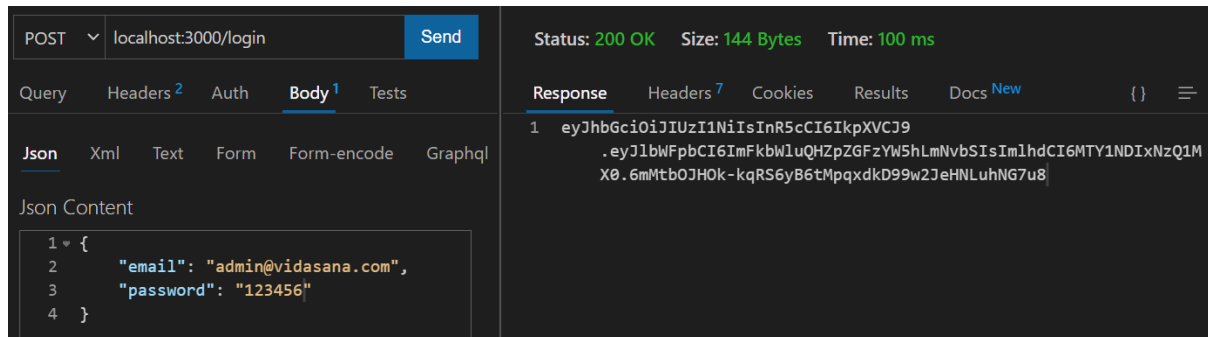


Imagen 9. Obteniendo un token como respuesta de una consulta
Fuente: Desafío Latam

Los tokens son una cadena alfanumérica (String) que es creada basado en un algoritmo predeterminado.

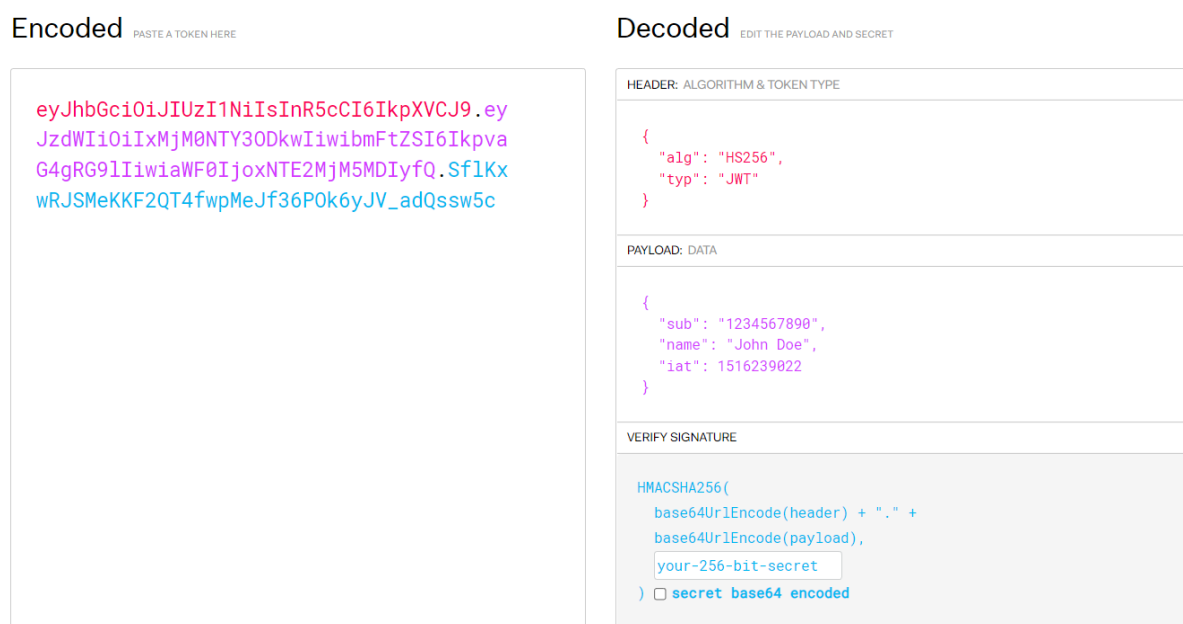


Imagen 10. Consola original de JWT
Fuente: jwt.io

JWT nos permite guardar un payload en su cifrado que puede ser obtenido al momento de decodificarlo

La composición de este tipo de token es la siguiente:



Imagen 11. Composición de un token
Fuente: Desafío Latam

Mientras que el flujo operacional que representa la consulta, autenticación, autorización y la comunicación con la base de datos es la siguiente:

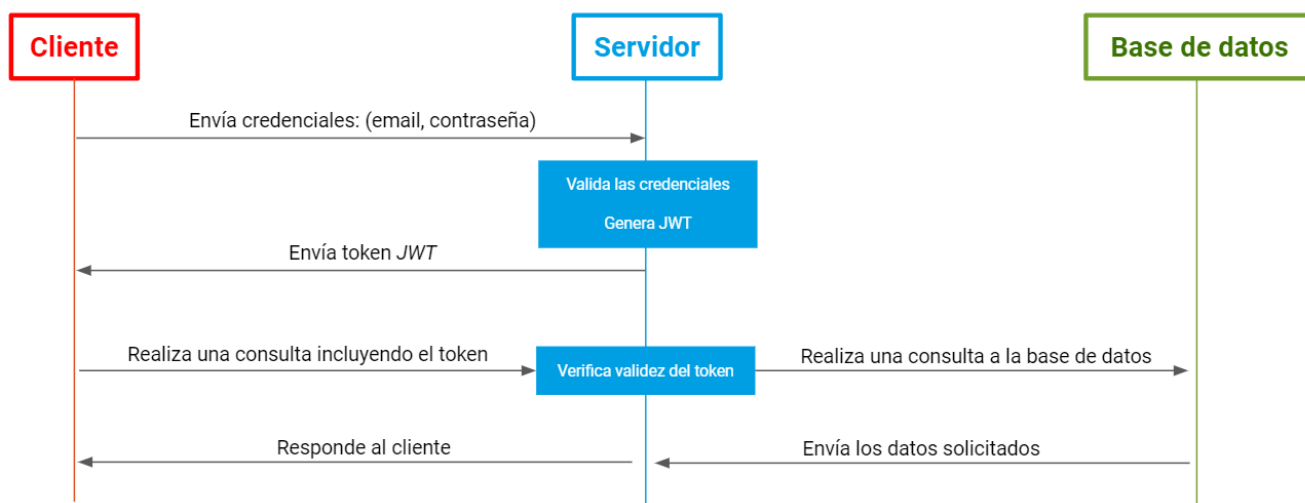


Imagen 12. Flujo operacional
Fuente: Desafío Latam



¡Manos a la obra! - Token con JWT

Utiliza las credenciales de otro usuario de la base de datos para obtener otro token. Luego entra al sitio web oficial de **JWT** <https://jwt.io/>

- Pega el token obtenido en la consola
- Observa el payload que contiene

- Modifica el payload
- Observa cómo cambia el token

Obteniendo un token en las cabeceras

Ahora que nuestras aplicaciones clientes pueden obtener tokens a partir de un inicio de sesión, usarán estos mismos tokens para solicitarnos información o funciones restringidas.

Creemos una ruta **DELETE /eventos/:id** que acceda a un **token** ubicado en las cabeceras de la consulta, específicamente en la propiedad **Authorization**:

```
app.delete("/eventos/:id", async (req, res) => {  
  try {  
    const { id } = req.params  
    const Authorization = req.header("Authorization")  
    const token = Authorization.split("Bearer ")[1]  
    console.log(token)  
  } catch (error) {  
    res.status(error.code || 500).send(error)  
  }  
})
```

Usando el método **headers** del objeto *request* accedemos al valor de la cabecera específicamente en el atributo *Authorization*.

Ahora realicemos una consulta a la ruta **DELETE /eventos/1** incluyendo en las cabeceras una propiedad **Authorization** y como valor la siguiente concatenación: **"Bearer <token>"**

Utiliza el token obtenido anteriormente para realizar este ejemplo

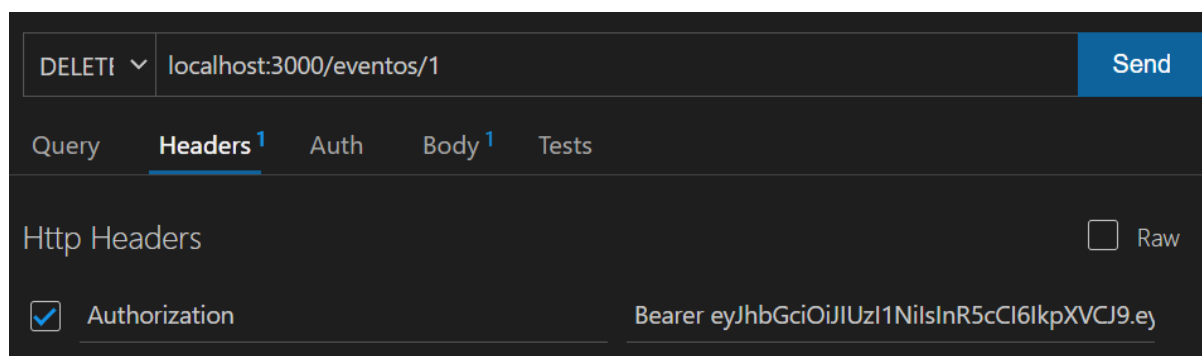


Imagen 13. Realizando una consulta con un token en las cabeceras
Fuente: Desafío Latam

Authorization: Bearer <token> proviene de un esquema de autenticación HTTP que involucra tokens de seguridad.

Al realizar la consulta anterior podremos revisar la terminal y verificar que estamos accediendo al token correctamente:

```
SERVER ON
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bWVpbCI6ImFkbWluQHZpZGFzYW5hLmNvbSI6Im1hdCI6MTY1NDIxNzQ1MX0.6mMtboJHOk-kqRS6yB6tMpqxdkD99w2JehNLuhNG7u8
```

Imagen 14. JWT mostrándose en la terminal
Fuente: Desafío Latam

Al visualizar esto sabremos que pudimos acceder correctamente al token de las cabeceras y podremos proceder con la verificación y decodificación del mismo.



¡Manos a la obra! - PUT /eventos/:id

Crea una ruta **PUT /eventos/:id** que:

- Muestre con console.log el **id** recibido en los parámetros
- Almacena en una variable el payload con los datos de un evento a actualizar
- Acceda y muestre por consola el token recibido en las cabeceras

Verificación y Decodificación de un JWT

En algunos casos necesitaremos acceder al payload dentro de un token JWT para realizar otras operaciones o simplemente incluirlos dentro de un mensaje o reporte.

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bWVpbCI6ImFkbWluQHZpZGFzYW5hLmNvbSI6Im1hdCI6MTY1NDIxNzQ1MX0.6mMtboJHOk-kqRS6yB6tMpqxdkD99w2JehNLuhNG7u8

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMCSHA256(
  base64urlEncode(header) + "." +
  base64urlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Obtendremos el payload de un token al decodificarlo

Imagen 15. El payload de un token en la consola JWT

Fuente: jwt.io

Continuando con la eliminación de un evento, procedamos con la verificación y decodificación del token recibido y el uso de la función **deleteEvento**

```
app.delete("/eventos/:id", async (req, res) => {
  try {
    const { id } = req.params
    const Authorization = req.header("Authorization")
    const token = Authorization.split("Bearer ")[1]
    jwt.verify(token, "az_AZ")
    const { email } = jwt.decode(token)
    await deleteEvento(id)
    res.send(`El usuario ${email} ha eliminado el evento de id ${id}`)
  } catch (error) {
    res.status(error.code || 500).send(error)
  }
})
```

La verificación de tokens se realiza por medio del método **verify**

```
jwt.verify(<token>, <llave secreta>)
```

Este método retorna un valor booleano y tiene integrada una excepción que podemos capturar usando el **try catch**

La decodificación de tokens se realiza por medio del método **decode**

```
const <payload> = jwt.decode( <token> )
```

La decodificación nos retorna directamente el payload utilizado en la firma de token.



El método **decode** no asegura que el token sea válido, solo descifra su información.

Intentemos eliminar el evento de id **2** a través de una consulta con Thunder Client

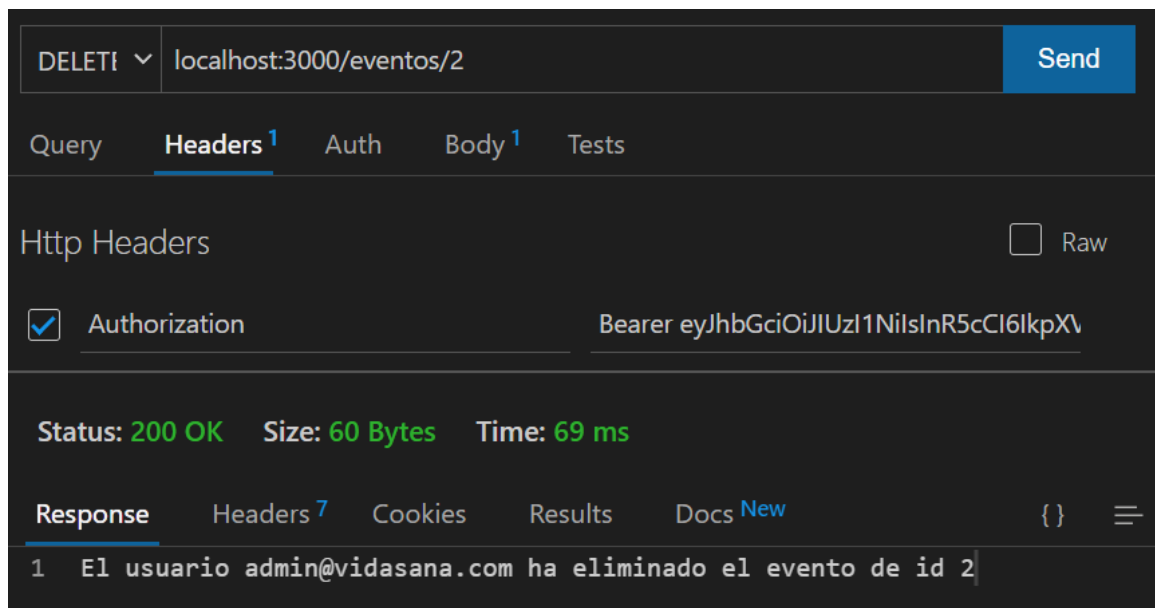


Imagen 16. Eliminando un evento usando un token para obtener autorización
Fuente: Desafío Latam



¡Manos a la obra! - Función actualizarEvento

Crea una función **actualizarEvento** que sea utilizada en la ruta **PUT /eventos/:id** luego de verificar y decodificar el token recibido

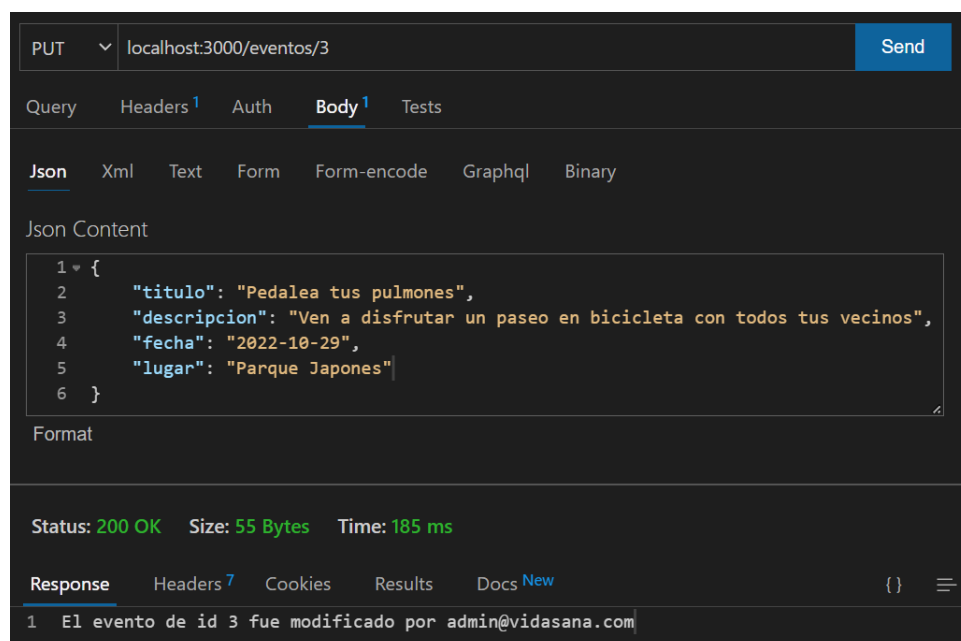


Imagen 17. Modificación de un evento usando en las cabeceras un token
Fuente: Desafío Latam

Cada vez que recibimos una consulta en nuestro servidor se está iniciando y finalizando un flujo de operaciones. Por lo que cada request es independiente de otro.

La comunicación nace y muere sin mantener un canal de comunicación entre el servidor y la aplicación cliente.

Encriptado de contraseñas

Una práctica frecuente en el desarrollo de un sistema de usuarios es el encriptado de las contraseñas, que tiene como objetivo preservar la privacidad de las credenciales de nuestros usuarios.

Para esto podemos instalar un paquete de npm llamado **bcryptjs**. Para instalarlo solo debes ejecutar la siguiente línea de comando:

```
npm install bcryptjs
```

El encriptado de contraseñas las usaremos en las mismas funciones en donde realizamos las consultas a la base de datos, por lo que deberás agregar la importación de este paquete en el archivo **consultas.js** de la siguiente manera:

```
const bcrypt = require('bcryptjs')
```

Registro de usuarios

Para integrar esta práctica a nuestro proyecto, crea y exporta una función **registrarUsuario** en **consultas.js**.

```
const registrarUsuario = async (usuario) => {  
  let { email, password } = usuario  
  const passwordEncriptada = bcrypt.hashSync(password)  
  password = passwordEncriptada  
  const values = [email, passwordEncriptada]  
  const consulta = "INSERT INTO usuarios values (DEFAULT, $1, $2)"  
  await pool.query(consulta, values)  
}
```

El método *hashSync* nos ayudará a encriptar la contraseña. Su anatomía es la siguiente:

```
bcrypt.hashSync( <password a encriptar> )
```

Al ejecutar esta línea siempre obtendremos una combinación de caracteres diferente generado por un algoritmo de criptografía que está integrado en el paquete *bcryptjs*.

Ahora agrega una nueva ruta en el servidor que utilice el método **POST** para recibir los datos de un nuevo usuario.

```
app.post("/usuarios", async (req, res) => {
  try {
    const usuario = req.body
    await registrarUsuario(usuario)
    res.send("Usuario creado con éxito")
  } catch (error) {
    res.status(500).send(error)
  }
})
```

Probemos esta ruta enviando las credenciales de un nuevo usuario:

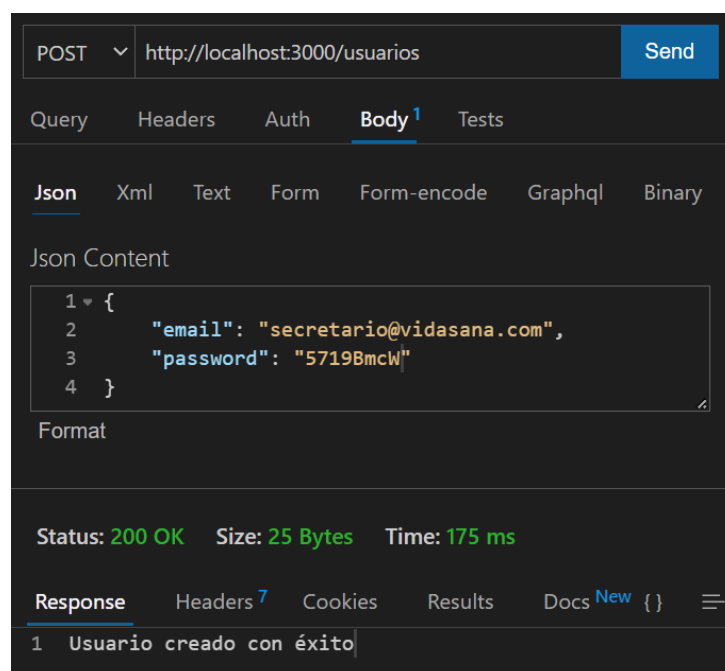


Imagen 18. Creando un nuevo usuario con contraseña encriptada
Fuente: Desafío Latam

Ahora, si revisamos la tabla usuarios desde la terminal *psql* podremos apreciar como se registró la contraseña encriptada y no la original.

```
vida_sana=# SELECT * FROM usuarios;
id | email | password
-----+-----+-----
 1 | admin@vidasana.com | 123456
 2 | manager@vidasana.com | abcdefg
 3 | secretario@vidasana.com | $2a$10$mFkDmtGckizluwaet63LujG8sJdWpJm9TZ0TxUb6achkYckI96Um
(3 filas)
```

Imagen 19. Verificando el registro de contraseña encriptada
Fuente: Desafío Latam

Con esta combinación de caracteres no podríamos maliciosamente hacernos pasar por el usuario que acabamos de registrar, no obstante si podríamos comparar una contraseña encriptada con su contraseña original y confirmar su fuente.

Inicio de sesión de usuarios

Para el proceso de login o inicio de sesión realizamos esta comparación al recibir la contraseña original escrita por el usuario y la contraseña encriptada que está registrada en la base de datos.

Modifiquemos la función **verificarCredenciales** para incluir la comparación de contraseñas en el proceso de verificación:

```
const verificarCredenciales = async (email, password) => {
  const values = [email]
  const consulta = "SELECT * FROM usuarios WHERE email = $1"

  const { rows: [usuario], rowCount } = await pool.query(consulta, values)

  const { password: passwordEncriptada } = usuario
  const passwordEsCorrecta = bcrypt.compareSync(password, passwordEncriptada)

  if (!passwordEsCorrecta || !rowCount)
    throw { code: 401, message: "Email o contraseña incorrecta" }
}
```

El método **compareSync** nos retorna **verdadero** o **falso** al realizar la comparación entre una contraseña encriptada y su supuesta contraseña original.

Intentemos iniciar sesión con las mismas credenciales del usuario que registramos y comprobemos que seguimos recibiendo un token como respuesta a pesar de que ahora haremos un login con un usuario cuya contraseña fue encriptada.

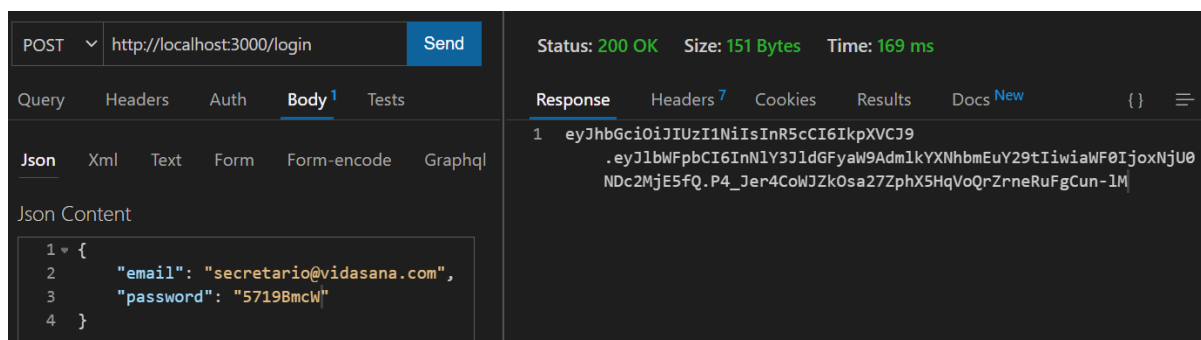


Imagen 20. Login con usuario cuya contraseña fue encriptada en su registro

Fuente: Desafío Latam



¡Manos a la obra! - Nuevo usuario

Registra un nuevo usuario en la base de datos, confirma que su contraseña fue encriptada y registrada en la tabla *usuario*, luego realiza el inicio de sesión usando las mismas credenciales que usaste para crearlo.

Límite de tiempo en token

JWT nos permite definir el tiempo de vida de un token expresado en milisegundos, segundos, horas e incluso días.

Para definir este tiempo solo se debe incluir un tercer argumento en la firma de un token:

```
const token = jwt.sign(<payload>, <llave secreta>, { expiresIn: <tiempo> } )
```

El valor que se le asigna al atributo ***expiresIn*** puede ser un número representado en segundos:

```
const token = jwt.sign(<payload>, <llave secreta>, { expiresIn: 60 } )
```

O también podemos definirlo con un *string* considerando la sintaxis de los siguientes ejemplos en donde se expresa un valor numérico acompañado de la unidad de tiempo, como por ejemplo:

```
const token = jwt.sign(<payload>, <llave secreta>, { expiresIn: "2 days" } )
```

```
const token = jwt.sign(<payload>, <llave secreta>, { expiresIn: "10h" } )
```

```
const token = jwt.sign(<payload>, <llave secreta>, { expiresIn: "7d" } )
```

Veamos un ejemplo modificando la ruta **POST /login** para agregar un tiempo límite del token de 60 segundos:

```
app.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body
    await verificarCredenciales(email, password)
    const token = jwt.sign({ email }, "az_AZ", { expiresIn: 60 })
    res.send(token)
  } catch (error) {
    console.log(error)
    res.status(error.code || 500).send(error)
  }
})
```

Ahora realicemos un inicio de sesión con el usuario de email **secretario@vidasana.com** y luego de 1 minuto, intentemos eliminar un evento a través de la ruta **DELETE /eventos/:id**:

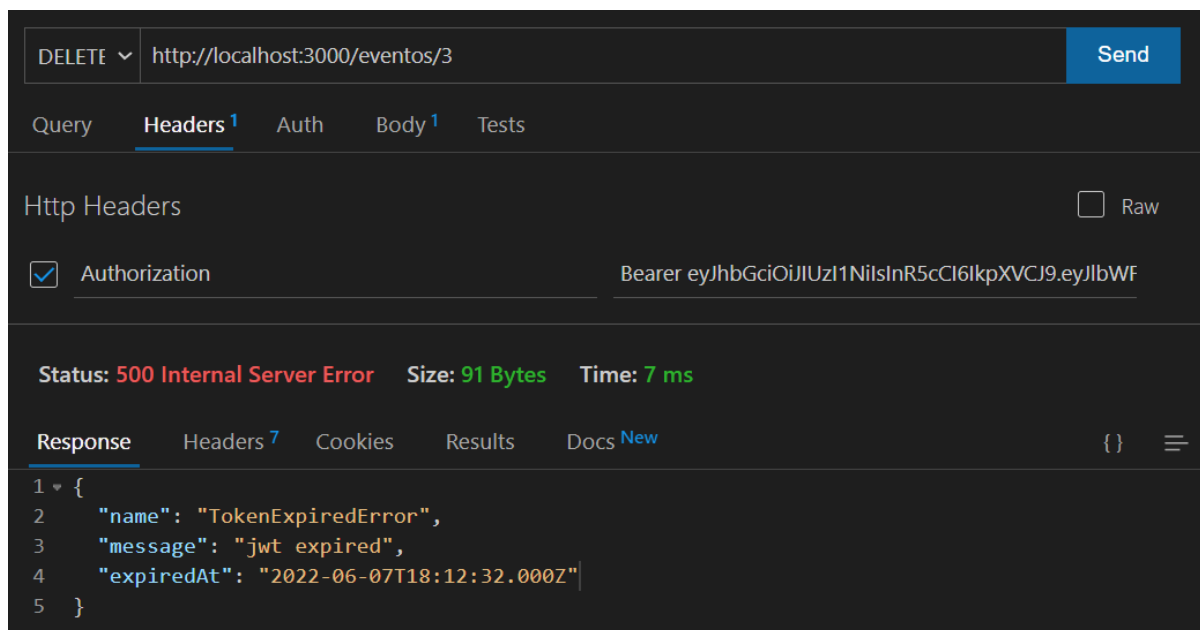


Imagen 21. Mensaje de token expirado
Fuente: Desafío Latam

Como podemos apreciar, nuestro servidor al intentar verificar el token y gracias a JWT detectó que el token recibido expiró.



¡Manos a la obra! - Nombre de la actividad

Modifica la ruta **POST /login** para que un token sea válido solo por 30 segundos, luego intenta iniciar sesión con un usuario y finalmente intenta actualizar un evento.

Deberás recibir como respuesta el mensaje que indica que el token expiró.

Preguntas:

- ¿Cuál es la diferencia entre autenticación y autorización?
- ¿Para qué sirve Json Web Token?
- ¿Cómo está compuesto un token de JWT?
- ¿En qué consiste el método verify?
- ¿En qué consiste el método decode?
- ¿Cuál es el objetivo del encriptado de contraseñas?
- ¿Cómo se puede definir el tiempo de vida de un token?