El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

CICLO IV:

Desarrollo de Aplicaciones Web

MisionTIC2022

El futuro digital es de todos | MinTIC

UNIVERSIDAD DEL NORTE
Vigilada Mineducación

Sesión 14:

Desarrollo de Aplicaciones Web

Desarrollo de Back-End web con Node.js

MisiónTIC2022

Objetivos de la sesión

Al finalizar esta sesión estarás en capacidad de:

1. Crear una aplicación que permita la gestión de la base de datos a través de un ORM.
2. Implementar conceptos base de seguridad para la seguridad de los usuarios.

Node.js - Uso de un ORM

- Por lo general para acceder a una base de datos se usa un package propio del software de la base de datos como los siguientes:
  - Mysql: mysql.
  - PostgreSQL: pg.
  - SQLite: sqlite3.
  - SQL Server: mssql.
  - Entre otros.
- Esto es engorroso ya que se nos resultaría muy difícil hacer un cambio de driver, es decir en vez de usar mysql, usar pg por ejemplo.
- Al proyecto también le haría falta conocer sobre la base de datos ya que no tendríamos información sobre la estructura de las tablas.

Node.js - Uso de un ORM

- Para agilizar este proceso de conexión y consulta a la base de datos por lo general se utiliza un Object Relational Mapping (ORM).
- El ORM nos facilita la consulta, lectura y conversión de los registros de una base de datos a objetos con lo que podemos trabajar.
- Un ORM nos permite definir la estructura de nuestra base de datos de una forma programática.
- Esto es posible ya que un ORM requiere hacer un esquema o modelo de datos tal cual como este estaría representado en la base de datos.
- También nos permite hacer migraciones de nuestros datos para hacer cambios sobre la estructura de la base de datos de una forma segura.

Node.js - Uso de un ORM

- Entre los ORM más populares para node, nos encontramos con los siguientes:
  - Sequelize.
  - TypeORM.
  - Prisma.
  - Entre otros.
- Estos ORM por lo general requieren un archivo de configuración e instalar el driver de la base de datos correspondiente.
- Un ORM nos permite cambiar de una base de datos de una forma sencilla sin requerir tantos cambios.

Node.js - ODM

- Similar a los ORM también existen los Object Document Mapping (ODM).
- Cabe resaltar que similar a un ORM, un ODM se encarga de hacer de intermediario entre la base de datos no relacional basada en documentos y nuestra aplicación.
- Entre los ODM más populares encontramos:
  - Mongoose.
  - TypeORM.
  - Entre otros.

Node.js - Mongoose

- Ya que la sesión anterior trabajamos con mongodb, en esta sesion estaremos adaptado lo ya desarrollado a mongoose, por lo que ejecutaremos **npm install mongoose** para agregarlos nuestro proyecto.
- Adicionalmente agregaremos una colección de usuarios y hablaremos sobre nociones de seguridad y manejo de sesiones o autenticación de usuarios.
- La base del proyecto la podemos encontrar en la rama main del siguiente repositorio.
- Así mismo los cambios realizados se encuentra en la rama using-mongoose del mismo repositorio,

Node.js - Mongoose Schema

- Mongoose introduce el concepto de schema, lo cual es la estructura que seguirán nuestros documentos, a continuación veremos un ejemplo de un esquema para usuarios:

```
const userSchema = new Schema(
  {
    user: String,
    password: String,
    tokenVersion: Number
  },
  {
    timestamps: { createdAt: 'createdAt', updatedAt: 'updatedAt' },
  }
);
```

- En este caso, la estructura de un usuario cuenta con los campos user, password y tokenVersion.
- Adicionalmente mongoose nos deja definir campos que serán manejados automaticamente como los timestamps createdAt y updatedAt.

Node.js - Mongoose Schema

- Se pueden agregar más configuraciones a un esquema:

```
const userSchema = new Schema(
  {
    user: {
      type: String,
      unique: true,
      required: [true, 'El usuario es requerido.'],
    },
    password: {
      type: String,
      select: false,
      required: [true, 'La contraseña es requerida.'],
    },
    tokenVersion: {
      type: Number,
      default: 0,
      required: false,
    },
  }, { timestamps: { createdAt: 'createdAt',
                     updatedAt: 'updatedAt' } });
```

Node.js - Mongoose Schema

- Como se puede ver podemos especificar lo siguiente:
    - El tipo del campo.
    - Si un campo es requerido.
    - El valor por defecto de un campo.
    - Si el campo ha de ser único.
- Adicionalmente, mongoose nos permite definir validadores para nuestros campos para verificar cosas como valor, longitud e incluso nos permite agregar validadores personalizados.

Node.js - Mongoose Schema

- A continuación extraemos un campo modificado de nuestro esquema para todos:

```
title: {
      type: String,
      validate: {
        validator: (v) => /^[ a-záéíóúñA-ZÁÉÍÓÚÑ0-9]*$/.test(v),
        message: 'El título no cumple con el formato especificado',
      },
      min: [3, 'El título ha de tener mínimo 3 caracteres'],
      max: [50, 'El título ha de tener máximo 50 caracteres'],
      required: [true, 'Título de la tarea requerido.'],
    },
```

- Como podemos ver en este caso validamos que el titulo del todo tenga mínimo 3 y máximo 50 caracteres.
- Adicionalmente si agregamos la validación en formato de vector podemos modificar el mensaje que mongoose maneja por defecto

Node.js - Mongoose Schema

- A continuación extraemos un campo modificado de nuestro esquema para todos:

```
title: {
      type: String,
      validate: {
        validator: (v) => /^[ a-záéíóúñA-ZÁÉÍÓÚÑ0-9]*$/.test(v),
        message: 'El título no cumple con el formato especificado',
      },
      min: [3, 'El título ha de tener mínimo 3 caracteres'],
      max: [50, 'El título ha de tener máximo 50 caracteres'],
      required: [true, 'Título de la tarea requerido.'],
    },
```

- Podemos definir una validación personalizada con validate la cual acepta las propiedades:
  - **validator:** espera un método que debe de retornar verdadero o falso.
  - **message:** espera una string que mongoose usará en caso de que el validator compute a falso.

Node.js - Mongoose Model

- Si mongoose define los Schema para referirse a los documentos, entonces Model es la forma en la que mongoose define o representa las colecciones de documentos.
- Siguiendo el ejemplo del user Schema podemos definir un modelo de la siguiente forma:

```
const User = model('User', userSchema);
```

- De esta forma mongoose define una forma con la que podemos interactuar con la base de datos de una forma sencilla.

Node.js - Mongoose Consultas

- Si fuéramos a hacer un endpoint para listar los usuarios entonces tendríamos que proceder de la siguiente forma:

```
export const getAllUsers = async () => await User.find();
```

- Si quisiéramos consultarlos por Id tendríamos que proceder de la siguiente forma:

```
export const getUserById = async (userId) => User.findById(userId);
```

- Si quisiéramos consultar filtrando por el valor de un campo podríamos hacer lo siguiente:

```
export const getUserByUser = async (user) => await User.findOne({ user: user });
```

- Para recuperar uno o más elementos usamos find y para uno usamos findOne
- Para más información ver la documentación de find de mongo y ver el uso de queries.

Node.js - Mongoose Escritura

- Si quisiéramos crear un usuario podemos proceder de la siguiente forma:

```
export const registerUser = async (payload) => {
  try {
    const newTodo = new User(payload);
    await newTodo.save();
    return newTodo;
  } catch (error) {
    if (error.code === 11000) throw new ClientError('Usuario no disponible');
    else throw error;
  }
};
```

- Creamos un documento a partir de nuestro modelo y el documento tiene un método para guardar cambios.
- Validamos que en caso de haber un error en el proceso no sea el código 11000.
- Este código representa que ya existe un usuario con ese user.

Node.js - Mongoose Editar & Eliminar

- Si quisiéramos actualizar un usuario, podemos proceder de la siguiente forma:

```
export const updateUser = async (userId, payload) =>
                                    await User.updateOne({ _id: userId }, payload);
```

- Donde payload sería la información final que quedará en la base de datos.
- Por otro lado para eliminar un usuario, podemos proceder de la siguiente forma:

```
export const deleteUser = async (userId) => await User.deleteOne({ _id: userId });
```

- Como podemos notar mongoose nos facilita interactuar con MongoDB.

Node.js - Nociones de seguridad

- Por otro lado, las aplicaciones han de tener nociones básicas de seguridad, como por ejemplo:
  - No almacenar la contraseña en texto plano.
  - Manejar sesiones con autenticación para mantener la información del usuario de forma constante.
  - Depurar valores introducidos en formularios por parte del usuario para evitar ataques como sql injection, etc.
  - Entre otros.

Node.js - Usar Funciones Hash

- Un podría usar el módulo crypto de node y hacer un hash a pedal, pero esto no es lo práctico debido a que hay paquetes que nos pueden facilitar el trabajo como bcrypt.

- Mongoose nos deja definir callbacks que se ejecutarán antes de ciertos eventos en nuestros schemas, razón por la cual añadiremos el siguiente callback usando bcrypt:

```
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  const salt = await genSalt(+process.env.BCRYPT_ROUNDS);
  this.password = await hash(this.password, salt);
  next();
});
```

- Cabe resaltar que genSalt y hash son métodos propios del paquete bcrypt.

Node.js - Usar Funciones Hash

- De igual forma sobreescribimos la contraseña con el hash.
- Como podemos notar no usamos las arrow functions.
- Esto se debe a que para acceder a la keyword this necesitamos usar function
- Lo cual nos comunica con el contexto en el cual corre el método y nos otorga acceso a la variable this.
- De esta forma evitamos incluir esta lógica en nuestro método de registro y solo modificamos el esquema.

El futuro digital es de todos | MinTIC

UNIVERSIDAD DEL NORTE
Vigilada Mineducación

Node.js - Autenticación de Usuarios

- Por lo general la autenticación se maneja de varias formas.
- Entre estas, las más populares son basadas en caché y basadas en tokens.
- Para la autenticación basada en caché, se maneja una base de datos a modo de caché como redis, donde se almacena información relacionada a cada usuario activo en la plataforma.
- Por otro lado la autenticación basada en tokens no requiere que el servidor maneje un estado de los usuarios ya que en cada solicitud se envía un token que autentica al usuario en el mismo.
- Para esta sesion estaremos implementando autenticación basada en tokens bajo el estándar de JsonWebTokens o JWT.

MisionTIC2022

Node.js - Json Web Tokens

- Primero instalaremos el paquete cookie-parser.
- Este paquete nos permitirá leer y agregar de una forma sencilla cookies a nuestras solicitudes y respuestas http.
- Luego instalaremos el paquete jsonwebtoken.
- Ya que este es el paquete que nos permitirá generar los tokens de sesión bajo el estándar y formato de JWT.

Node.js - Json Web Tokens

- Luego en **src/index.js** agregamos el middleware de cookieParser justo después del de compression de la siguiente forma:

```
app.use(compression());
app.use(cookieParser());
```

- Luego definimos un controlador para autenticación y un método para hacer inicio de sesión.
- Para agregar el controlador nos vamos a **src/controllers/index.js** y agregamos la siguiente línea dentro del método setUpControllers:

```
app.use('/auth', AuthRouter);
```

Node.js - Json Web Tokens

- Para agregar el endpoint de inicio de sesion creamos el archivo **src/controllers/auth/index.js** y definimos la ruta '/login':

```
router.post('/login', async (req, res, next) => {
  try {
    const { refreshToken, accessToken } = await loginUser(req.body);
    res
      .cookie('refreshToken', refreshToken, { httpOnly: true })
      .json({ message: 'El usuario ha inicado sesión', accessToken });
  } catch (error) {
    next(error);
  }
});
```

- Esta estrategia de inicio de sesión consta en tener 2 tokens, uno de acceso y otro para refrescar el de acceso.

Node.js - Json Web Tokens

- Entonces el token de acceso va únicamente en los cookies de la respuesta y el navegador web lo usará para todas las solicitudes que el cliente enviará en un futuro puesto que estamos indicando la opción de httpOnly como verdadera.
- Por lo general no se recomienda utilizar nombres sencillos de entender para las cookies como lo es el caso de este ejemplo con el nombre de ‘refreshToken’, ya que en estos tokens podría haber información sensible.
- El token de acceso lo enviamos en el cuerpo de nuestra respuesta.

Node.js - Json Web Tokens

- Para ver el detalle de cómo hacemos el inicio de sesión veremos el método loginUser proveniente del archivo src/controllers/auth/methods.js:

```
export const loginUser = async (payload) => {
  const user = await User.findOne({ user: payload.user });
  if (!user) throw new NotFoundError('Usuario no encontrado.');

  const passwordMatch = await user.comparePassword(payload.password);
  if (!passwordMatch) throw new ClientError('Contraseña invalida.');

  return await getTokenPair(user);
};
```

- Donde en primer lugar verificamos si existe el usuario y en segundo lugar probamos si la contraseña introducida es válida con el hash que tenemos almacenado en base de datos.

Node.js - Json Web Tokens

- Para comparar las contraseñas definimos un método personalizado en nuestro userSchema de mongo, de esta forma todos los documentos de la colección User tendrán acceso a él:

```
userSchema.methods.comparePassword = async function (plaintext) {
  return await compare(plaintext, this.password);
};
```

- Cabe resaltar que el método compare proviene del paquete bcrypt.
- Luego generamos el par de tokens de acceso y refresco con el método getTokenPair.

Node.js - Json Web Tokens

- Los tokens los generamos de la siguiente forma

```
const getTokenPair = async (user) => {
  const accessToken = await sign(
    {
      user: { _id: user._id, tokenVersion: user.tokenVersion, user: user.user },
    },
    process.env.JWT_ACCESS_SECRET,
    { expiresIn: '5m' }
  );

  const refreshToken = await sign(
    { user: { _id: user._id, tokenVersion: user.tokenVersion } },
    process.env.JWT_REFRESH_SECRET,
    { expiresIn: '7d' }
  );

  return { refreshToken, accessToken };
};
```

Node.js - Json Web Tokens

- Cabe resaltar que el método sign proviene del paquete jsonwebtoken. Y este acepta 3 parámetros:
    - La carga útil, nunca se incluye la contraseña por motivos de seguridad.
    - El secreto o llave privada, solo la debe de poseer el servidor.
    - La configuración.
- Es importante tomar en cuenta que los secretos o llaves privadas usadas para generar cada par de tokens es la misma pero entre token de acceso y de refresco no es el mismo.

Node.js - Json Web Tokens

- La carga útil es diferente, ya que el token de refresco solo tiene información básica para encontrar el usuario, mientras que el de acceso contiene más datos como el correo, etc.
- El token de acceso tiene una duración corta, a lo mucho 5 minutos.
- El token de refresco tiene una duración larga puede ser semanas, meses, años, etc.

Node.js - Autorización

- Una vez el cliente posee ambos tokens y es capaz de enviar la solicitud con el encabezado de autorización (En este caso nosotros trabajaremos con el Bearer token), esto nos permite del lado del servidor implementar rutas ‘seguras’.
- Una ruta segura es una ruta que se encarga de validar que el token que venga en el encabezado de autorización haya sido generado por el servidor y sea válido.
- Para esto nos apoyaremos en la versatilidad que nos ofrece Express para agregar middlewares o funciones intermedias para enriquecer nuestras solicitudes.

Node.js - Autorización

- Para esto crearemos el archivo **src/guards/auth.js**, y definiremos el siguiente método:

```
export const authGuard = (req, _res, next) => {
  const authorization = req.headers.authorization;

  if (!authorization) throw new UnAuthorizedError('No se encuentra autenticado');

  try {
    const token = authorization.split(' ')[1];
    const payload = verify(token, process.env.JWT_ACCESS_SECRET);
    req.jwt_payload = payload;
  } catch (err) {
    console.log(err);
    throw new UnAuthorizedError('Fallo la verificación del token');
  }

  return next();
};
```

Node.js - Autorización

- Cabe resaltar que el método verify proviene de jsonwebtoken.
- Por otro lado la carga útil se la agregaremos a la solicitud en caso de necesitarla en alguno de nuestros métodos.
- Si la autenticación falla le enviaremos al cliente una respuesta con un código de respuesta 401 para indicar que el usuario no se encuentra autorizado o que su sesión expiró.
- Si quisiéramos utilizar este guard, tendríamos que proceder de la siguiente forma:

```
router.get('/list', authGuard, async (_req, res, next) => {
  try {
    const todos = await getAllTodos();
    res.json(todos);
  } catch (error) {
    next(error);
  }
});
```

- Este es el endpoint '/to-do/list'.

Node.js - Refrescar Token de Acceso

- Ya que la vida útil del token de acceso es muy corta (menos de 5 minutos), vamos a tener que estar actualizando constantemente.
- Para esto definimos un método para refrescar el token en nuestro AuthController.

```
router.post('/refresh', async (req, res, next) => {
  try {
    const { refreshToken, accessToken } = await
refreshAccessToken(req.cookies.refreshToken);
    res
      .cookie('refreshToken', refreshToken, { httpOnly: true })
      .json({ message: 'Token de acceso actualizado', accessToken });
  } catch (error) {
    next(error);
  }
});
```

- Como podemos ver actualizamos tanto el token de acceso como de refresco, esto aumenta el tiempo de sesión del usuario, por lo general es muy usado en redes sociales.

Node.js - Refrescar Token de Acceso

- Por último, el método para actualizar el token es:

```
export const refreshAccessToken = async (refreshToken) => {
  if (!refreshToken)
    throw new NotFoundError('No se encontro el refresh token.');

  const payload = verify(refreshToken, process.env.JWT_REFRESH_SECRET);
  const user = await User.findById(payload.user._id);

  if (!user) throw new NotFoundError('No se encontro el usuario.');

  if (user.tokenVersion !== payload.user.tokenVersion)
    throw new TokenExpiredError('Este token no es valido.');

  return await getTokenPair(user);
};
```

- Donde validamos lo mismo, el usuario y si el token es válido.
- Adicionalmente también versionamos nuestros tokens.

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

Ejercicios de práctica

Mision TIC 2022

El futuro digital es de todos MinTIC

UNIVERSIDAD DEL NORTE
Vigilada Mineducación

Referencias

- https://www.section.io/engineering-education/how-to-build-authentication-api-with-jwt-token-in-nodejs/
- https://scoutapm.com/blog/express-error-handling
- https://www.thepolyglotdeveloper.com/2019/02/hash-password-data-mongodb-mongoose-bcrypt/
- https://techbrij.com/token-authentication-nodejs-express-mongo-passport

Mision TIC 2022

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

¡GRACIAS

POR SER PARTE DE ESTA EXPERIENCIA DE APRENDIZAJE!

Mision TIC 2022