



API REST - JWT

Para implementar paso a paso nuestra **API REST** con **Nodejs + Express + MongoDB** necesitamos definir unos requisitos y dependencias del proyecto

Introducción:

Para empezar vamos a ir definiendo el objetivo de la api, los modelos que va a tener, etc. El objetivo de la API es crear paletas de colores en combinaciones de 4 u 8 colores.

Para ello la API se va a apoyar en 3 colecciones de MongoDB (User, Palette, Color). La colección User va a contener la información de los usuarios, la colección Color va a contener una lista de colores con las que se llenaran la paletas y por último la colección Palette va a tener una relación con la colección de colores y otra con la colección de usuarios.





Las peticiones que va a recibir nuestra Api son de lectura para colores, un CRUD completo para las paletas y para los usuarios un registro y un login. La seguridad de la aplicación va a ser mediante JWT y utilizaremos un archivo para gestionar los status code de las peticiones.

¡Empezamos 🙌!

Guía de trabajo:

Antes de comenzar haremos un breve repaso de las herramientas que necesitamos tener en nuestro equipo y os dejaremos el link de cada uno de los paquetes con los que vamos a trabajar:

Setup → Herramientas en de tu equipo para poder seguir esta guía:




-  [NodeJs](#)
-  [Npm](#)
-  [MongoDB](#)
-  [Insomnia](#)

Dependencies → Dependencias, librerías que nos ayudarán con el desarrollo y funcionalidades de nuestra aplicación:

-  [Express](#)
-  [Mongoose](#)

-  Cors
-  Bcrypt
-  Jsonwebtoken

Dev-Dependencies → Dependencias que nos ayudan en el desarrollo de nuestra aplicación:

-  Nodemon
-  Morgan
-  Dotenv

Creando un proyecto

Iniciamos un proyecto con node. Para ello lo primero es crear una carpeta con el nombre de proyecto, una vez dentro crear una carpeta server, que es el sitio donde se aloja nuestra API, ya en server ejecutaremos el `init -y` para crear el `package.json`:

```
mkdir name-project
cd name-project
mkdir server
cd server
npm init -y
```

Instalando dependencias

Express: Nuestra librería principal para la configuración de nuestro servidor.

```
npm i express
```

Mongoose: Utilizaremos mongoose para comunicarnos con la base de datos de MongoDB

```
npm i mongoose
```

Cors: Utilizaremos cors para determinar desde donde podrán acceder a nuestra api y también configura las cabeceras de nuestra API REST de forma automática.

```
npm i cors
```

Bcrypt: Nos permitirá cifrar las contraseñas.

```
npm i bcrypt
```

Jsonwebtoken: Librería que mas adelante utilizaremos para gestionar los tokens.

```
npm i jsonwebtoken
```

Nodemon: Nodemon es una herramienta de desarrollo la cual se encarga de levantar el servidor cada vez que realizamos cambios y así evitar estar tirando y levantando el servidor.

```
npm i nodemon -D
```

Morgan: Herramienta de desarrollo que nos imprime las peticiones que se van recibiendo en la API.

```
npm i morgan -D
```

Dotenv: Nos permite consumir las variables de entorno.

```
npm i dotenv -D
```

Scripts de arranque

En el archivo `package.json` podemos comprobar que tenemos instaladas todas las dependencias. Además debemos introducir los siguientes scripts:

```
"scripts": {  
  "start": "node server.js",  
  "dev": "nodemon server.js"  
}
```

El script `start` levanta el servidor de forma manual. Para ello escribimos en la terminal el comando `npm run start`

El script `dev` ejecuta nodemon que se encargará de tumbar/levantar nuestro servidor. Para ello escribimos en la terminal el comando `npm run dev`

Estructura del proyecto

Esta es la estructura que vamos a seguir durante el proyecto.

Vamos a ir comentando carpeta por carpeta los archivos que contendrá.

- app : Esta carpeta será nuestra carpeta principal donde iremos añadiendo las necesarias.
 - api: Aquí iremos añadiendo las carpetas necesarias para gestionar nuestras rutas y ejecutarán la lógica.
 - models: Contendrá los modelos de mongoose
 - User.js
 - Palettes.js
 - Color.js).
 - routes: Contendrá los archivos que se encargaran de gestionar las peticiones y llamar a las funciones de los controladores para que efectúen la lógica que queramos.
 - user.routes.js
 - palette.routes.js
 - color.routes.js
 - controllers: Contiene los archivos que se encargarán de ejecutar la lógica de nuestra aplicación.
 - user.controller.js

- palette.controller.js
- color.controller.js
- config: Contendrá los archivos de configuración, en nuestro caso contendrá el archivo de configuración con la base de datos.
 - database.js
- middlewares: Contendrá los middlewares de nuestra API.
 - auth.middleware.js
- utils: Contiene la utilidades como su nombre indica. en nuestro proyecto tendrá un archivo que contiene los mensajes de estado de las peticiones.
 - httpStatusCode.js
- server.js: Es el punto de entrada de nuestro servidor, aquí configuraremos express.
- .env: Contiene nuestras variables de entorno

```

└─ app
  └─ api
    └─ models
      └─ User.js
      └─ Palette.js
      └─ Color.js
    └─ routes
      └─ user.routes.js
      └─ palette.routes.js
      └─ color.routes.js
    └─ controllers
      └─ user.controller.js
      └─ palette.controller.js
      └─ color.controller.js
    └─ seeds
      └─ color.seed.js
    └─ config
      └─ database.js
    └─ middlewares
      └─ auth.middleware.js
    └─ utils
      └─ httpStatusCode.js
  └─ server.js
  └─ .env

```

Configuración de las variables de entorno

Nuestras variables de entorno las definiremos en el archivo **.env** que crearemos en la raíz de nuestro proyecto.

Esta es nuestra variable que contiene la url de Mongo Atlas. Mas tarde utilizando **dotenv** la podremos consumir.

```
MONGO_DB=mongodb+srv://<user>:<password>@cluster0.3thlc.mongodb.net/<Nombre de la db>?retryWrites=true&w=majority
```

Levantando el servidor

Para comenzar a utilizar **Express** en nuestro proyecto iremos a nuestro archivo **server.js** y lo configuramos de la siguiente manera.

```

const express = require("express");
const app = express();

app.listen(3000, () => {

```

```
console.log("Node server listening on port 3000");
};
```

Comprobamos si todo está bien lanzado de nuevo el comando `npm run dev`. Con ayuda de express ya tenemos levantado el servidor, es decir, ya podemos recibir peticiones en <http://localhost:3000>. De momento no lo hemos configurado todavía, pero ya esta a la "escucha".

Conexión a MongoDB

En esta ocasión utilizaremos **MongoDB Atlas**, pero se podría hacer con **MongoDB** en local. Para conectar con nuestra base de datos **MongoDB** usaremos la librería **Mongoose** que permite modelar los datos de Mongo en objetos de Javascript.

Vamos a crear un archivo `database.js` para la configuración de la conexión a nuestra base de datos. Este archivo realizará la conexión con la base de datos y la mantendrá abierta para poder comunicarnos con ella.

```
//Requerimos dotenv para acceder a las variables de entorno
const dotenv = require("dotenv");
dotenv.config();
//Requerimos mongoose para comunicarnos con la bd
const mongoose = require("mongoose");
//guardamos la url de Mongo en una variable
const mongoDb = process.env.MONGO_DB;
//Configuramos la función connect
const connect = async () => {
  try {
    const db = await mongoose.connect(mongoDb, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    const { name, host } = db.connection;
    console.log(`Connected with db: ${name}, in host: ${host}`);
  } catch (error) {
    console.log("Error to connect with BD", error);
  }
};
//exportamos la funcion connect
module.exports = { connect };
```

Importamos la función Connect en `server.js` y la ejecutamos.

```
//Imports
const express = require("express");
//Importamos la conexion a la db
const {connect} = require("../app/config/database");

//Ejecutamos la funcion que conecta con la db
connect();

const app = express();

//Config app

app.listen(3000, () => {
  console.log("Node server listening on port 3000");
});
```

En este punto si ejecutamos `npm run dev` debemos de ver que tanto nuestro server como la conexión a la base de datos ha ido bien.



Configurando nuestro servidor

Para realizar la configuración de nuestro servidor vamos a configurar el archivo `server.js`

En esta primera parte de la configuración estamos definiendo los **headers** de nuestra respuesta (res). En cada petición express se encargará de crear la cabecera con la configuración que hemos determinado.

```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE');
  res.header('Access-Control-Allow-Credentials', true);
  res.header('Access-Control-Allow-Headers', 'Content-Type');
  next();
});
```

En este paso añadimos **cors** y definimos las direcciones que van a tener permiso para utilizar nuestra API. De momento en local:

```
const cors = require("cors");

app.use(cors({
  origin: ['http://localhost:3000', 'http://localhost:4200'],
  credentials: true,
}));
```

Con estas dos líneas de código configuramos express para poder enviar y recibir información en el **body** de las peticiones en formato JSON.

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Importamos la dependencia **morgan** y la configuramos en el servidor. Esto nos ayudará a ver las peticiones que estamos lanzando.

```
const logger = require("morgan");

app.use(logger("dev"));
```

Realizamos las importaciones de los archivos que contendrán los "gestores" de las rutas. Como vemos tendremos uno por cada colección de la base de datos. Cuando recibamos una petición de cualquier tipo a la dirección `localhost:3000/colors` express analizará la ruta y al encontrar `/colors` directamente llamará a `color.routes.js` que será el encargado de ver que tipo de petición se esta recibiendo y llamará a la función del controlador responsable de hacer la lógica.

```
const users = require("./app/api/routes/user.routes");
const colors = require("./app/api/routes/color.routes");
const palettes = require("./app/api/routes/palette.routes");

// routes
app.use("/users", users);
app.use("/colors", colors);
app.use("/palettes", palettes);
```

Con estas líneas de código estamos controlando las rutas que no coincidan con ninguna definida en el punto anterior. Por ejemplo si recibimos una petición a `localhost:3000/paint`, express va a realizar una comprobación con las rutas definidas y como no hay coincidencia entra en esta función y ejecuta un error con el status 404 y con el mensaje "Not Found".

```
const HTTPSTATUSCODE = require("./app/utls/httpStatusCode");

app.use((req, res, next) => {
  let err = new Error();
  err.status = 404;
  err.message = HTTPSTATUSCODE[404];
  next(err);
});
```

Para gestionar los mensajes de error de nuestra api nos vamos a apoyar en un archivo de utilidad denominado `httpStatusCode.js` que esta alojado en el directorio `utls`. El contenido del archivo esta disponible en el siguiente enlace:

 [httpStatusCode.js](#)

Este va a ser nuestro controlador de errores. Cuando a lo largo de nuestra aplicación se produzca un error se llamará a la función `next` y se le pasará el error como argumento. Esta función es la que se va a encargar de recibir el error y devolverlo en un json.

```
// handle errors
app.use((err, req, res, next) => {
  return res.status(err.status || 500).json(err.message || 'Unexpected error');
})
```

Para evitar que se sepa que nuestra api rest está montada con Node.js escribimos `app.disable('x-powered-by')`, ya que Express añade esta información en cada petición.

```
app.disable('x-powered-by');
```

Finalmente nuestro fichero `server.js` quedaría:

```
//Imports
const express = require("express");
const logger = require("morgan");
//Importamos la conexion a la db
const {connect} = require("./app/config/database");
//Importamos las rutas
const users = require("./app/api/routes/user.routes");
const colors = require("./app/api/routes/color.routes");
const palettes = require("./app/api/routes/palette.routes");
//Otras importaciones
const HTTPSTATUSCODE = require("./app/utls/httpStatusCode");
const cors = require("cors");

//Conectamos con la db
connect();

const app = express();
//Config app

app.use((req, res, next) => {
  res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE');
  res.header('Access-Control-Allow-Credentials', true);
  res.header('Access-Control-Allow-Headers', 'Content-Type');
  next();
});

app.use(cors({
  origin: ['http://localhost:3000', 'http://localhost:4200'],
  credentials: true,
}));
```

```

app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.use(logger("dev"));

// routes
app.use("/users", users);
app.use("/colors", colors);
app.use("/palettes", palettes);

app.use((req, res, next) => {
  let err = new Error();
  err.status = 404;
  err.message = HTTPSTATUSCODE[404];
  next(err);
});

// handle errors
app.use((err, req, res, next) => {
  return res.status(err.status || 500).json(err.message || 'Unexpected error');
})

app.disable('x-powered-by');
//Levantamos el servidor
app.listen(3000, () => {
  console.log("Node server listening on port 3000");
});

```

Definimos los modelos de nuestra DB

El siguiente paso para configurar nuestra API es crear los modelos con la dependencia **mongoose**. Con los modelos vamos a conseguir que nuestras colecciones en base de datos tengan una estructura.

Para ello vamos a empezar a definir el modelo `Color.js`

```

// Cargamos el módulo de mongoose
const mongoose = require("mongoose");
// Definimos los esquemas
const Schema = mongoose.Schema;
// Creamos el objeto del esquema con sus correspondientes campos
const ColorSchema = new Schema(
  {
    hex: { type: String, require: true },
    name: { type: String, require: true },
    rgb: { type: String, require: true },
  },
  { timestamps: true }
);
// Exportamos el modelo para usarlo en otros ficheros
const Color = mongoose.model("colors", ColorSchema);
module.exports = Color;

```

Como podemos ver en el código el modelo color va a tener tres propiedades obligatorias que son:

1. **hex**: Contendrá el código de color en valor hexadecimal.
2. **name**: Contendrá el nombre del color.
3. **rgb**: Contendrá el código de color en valor rgb.
4. **timestamps**: Es una propiedad que igualandola a true, conseguiremos que nos guarde un valor con la fecha de creación y de modificación del documento.

Ya tenemos definido el modelo `Color.js`, ahora definiremos el modelo de Palette y de User. `Palette.js`


```
// Cargamos el módulo de mongoose
const mongoose = require("mongoose");
// Definimos los esquemas
const Schema = mongoose.Schema;
// Creamos el objeto del esquema con sus correspondientes campos
const PaletteSchema = new Schema(
  {
    name: { type: String, required: true },
    description: { type: String, required: true },
    //relacionamos la propiedad colors con la colección colors
    colors: [{ type: Schema.Types.ObjectId, ref: "colors", required: true }],
    //relacionamos la propiedad author con la colección users
    author: { type: Schema.Types.ObjectId, ref: "users", required: true }
  },
  { timestamps: true }
);

const Palette = mongoose.model("palettes", PaletteSchema);
module.exports = Palette;
```

Destacaremos de este modelo la relación entre diferentes modelos, como podemos observar en el código, la propiedad colors es un array de objetos los cuales hacen referencia a la colección colors y la propiedad author hace referencia a la colección de users que vamos a crear a continuación. Por lo tanto cuando creamos una nueva paleta deberemos introducir los ids de los colores que queramos guardar y el id de author..

Vamos ahora con el modelo de usuario, `User.js`

```
// Cargamos el módulo de mongoose
const mongoose = require("mongoose");
// Cargamos el módulo de bcrypt
const bcrypt = require("bcrypt");
// Definimos el factor de costo, el cual controla cuánto tiempo se necesita para calcular un solo hash de BCrypt. Cuanto mayor sea el facto
const saltRounds = 10;
// Definimos los esquemas
const Schema = mongoose.Schema;
// Creamos el objeto del esquema con sus correspondientes campos
const UserSchema = new Schema({
  name: { type: String, trim: true, required: true },
  emoji: { type: String, trim: true, required: true },
  email: { type: String, trim: true, required: true },
  password: { type: String, trim: true, required: true },
  favPalettes: [{ type: Schema.Types.ObjectId, ref: "palettes" }],
});
// Antes de almacenar la contraseña en la base de datos la encriptamos con Bcrypt
UserSchema.pre("save", function (next) {
  this.password = bcrypt.hashSync(this.password, saltRounds);
  next();
});
// Exportamos el modelo para usarlo en otros ficheros
const User = mongoose.model("users", UserSchema);
module.exports = User;
```

El modelo User.js va a tener las propiedades determinadas en el Schema, al igual que en los modelos anteriores. La diferencia con los modelos anteriores es que antes de exportar el usuario se realiza un encriptado de la contraseña, con la ayuda de la función `hashSync` de la dependencia bcrypt.

👉 ¡¡Ya tenemos los modelos creados!! Como vemos nuestra API ya va cogiendo forma.

Creación de una semilla de colores

Ya tenemos creado los modelos pero si visitamos nuestra colección en MongoDB Atlas vamos a ver que esta vacía. La colección de colores es solo de lectura y va a contener un listado de colores, por este motivo vamos a introducir en nuestra colección un montón de colores de golpe. Para ello nos vamos a crear una semilla con el nombre `color.seed.js` ubicada en `seeds` dentro de `api`.

```

const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config();

// Importamos el modelo Color.js en este nuevo archivo.
const Color = require('../models/Color');

const colors = [ ... ];

// En este caso, nos conectaremos de nuevo a nuestra base de datos
// pero nos desconectaremos tras insertar los documentos
mongoose
  .connect(process.env.MONGO_DB, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(async () => {
    // Utilizando Color.find() obtendremos un array con todos los colores de la db
    const allColors = await Color.find();

    // Si existen colores previamente, dropearemos la colección
    if (allColors.length) {
      await Color.collection.drop(); //La función drop borra la colección
      console.log('Drop database')
    }
  })
  .catch((err) => console.log(`Error deleting data: ${err}`))
  .then(async () => {
    // Una vez vaciada la db de los colores, usaremos el array colors
    // para llenar nuestra base de datos con todas los colores.
    await Color.insertMany(colors);
    console.log('DatabaseCreated')
  })
  .catch((err) => console.log(`Error creating data: ${err}`))
  // Por último nos desconectaremos de la DB.
  .finally(() => mongoose.disconnect());

```

Para no hacer el código muy largo, el valor de la variable `colors` se puede obtener del siguiente enlace:

 [colors](#)

Una vez creado la semilla la ejecutaremos introduciendo el siguiente comando en la terminal.

```
node app/api/seeds/color.seed.js
```

Si todo ha ido bien debería aparecer en consola el texto: **DatabaseCreated**

Ahora que ya tenemos documentos en la colección de colores vamos a empezar a definir nuestros "gestores" de rutas. Como vimos en la configuración del archivo `server.js` cuando entraba una petición a `localhost:3000/colors` redireccionabamos la petición al enrutador `color.routes.js`. Pues bien vamos a configurar nuestros archivos `color.routes.js`, `palette.routes.js` y `user.routes.js`

Configurando el routing

Vamos a comenzar a configurar nuestro archivo `color.routes.js`.

```

//Importamos express
const express = require("express");
//Guardamos la funcion express.Router() en una variable
const router = express.Router();

//Importamos las funciones del controlador de color
const { getAllColors, getColorById } = require("../controllers/color.controller");
//Definimos el metodo, la ruta de entrada y la función del controlador
//que se encargará de efectuar la lógica
router.get("/", getAllColors);

```

```
router.get("/:colorId", getColorById);
//exportamos la variable router
module.exports = router;
```

Vamos a explicar un poco como actuará el código. supongamos que recibimos una petición GET a localhost:3000/colors, esta petición entrará en el server, el cual "troceará" la ruta y se quedará con `/colors`, como hay coincidencia llamará a `color.router.js`. Éste archivo comprobará el método, en este caso GET, y la ruta que queda, en este caso `('/')`, como hay coincidencia llamará a la función `getAllColors` que tendrá que conectarse con la bd y devolver todos los colores.

Te habrás dado cuenta que estamos importando las funciones `getAllColors` y `getColorById` pero éstas todavía no las hemos creado, ese va a ser nuestro siguiente paso, crear en `color.controller.js` estas funciones y exportarlas para que las pueda consumir `color.routes.js`.

En este punto de la construcción de nuestra API vamos a continuar con la configuración del controlador de colores. Dejamos en "stand by" el routing de paletas y de usuarios, mas tarde lo retomaremos.

Configurando controladores

Los controladores van a tener la función de realizar toda la lógica con la base de datos. Vamos a comenzar por configurar nuestro archivo `color.controller.js`.

```
// Cargamos el modelo recién creado
const Color = require("../models/Color");
// Cargamos el fichero de los HTTPSTATUSCODE
const HTTPSTATUSCODE = require("../utils/httpStatusCode");

//Metodo para retornar todos los colores registrados en la base de datos

const getAllColors = async (req, res, next) => {
  try {
    if (req.query.page) { //Se le añade paginación
      const page = parseInt(req.query.page);
      const skip = (page - 1) * 20;
      const colors = await Color.find().skip(skip).limit(20);
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { colors: colors },
      });
    } else {
      const colors = await Color.find();
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { colors: colors },
      });
    }
  } catch (err) {
    return next(err);
  }
};

// Metodo para la búsqueda de colores por ID
const getColorById = async (req, res, next) => {
  try {
    const { colorId } = req.params;
    const colorById = await Color.findById(colorId);
    return res.json({
      status: 200,
      message: HTTPSTATUSCODE[200],
      data: { colors: colorById }
    });
  } catch (err) {
    return next(err);
  }
};

//Exportamos las funciones
module.exports = {
```

```

    getAllColors,
    getColorById,
  }
}

```

Como podemos observar en el código hemos creado dos funciones una denominada `getAllColors` que tiene la función de conectarse con bd y traer todos los documentos de la colección Color y también tiene la función `getColorById` que se encargará de recoger de la ruta el id de un color determinado y realizar una búsqueda en la bd para traer la información del color que tenga el mismo id. Para terminar las exportamos para que puedan ser consumidas por `color.routes.js`.

En este punto podemos realizar una petición desde el navegador (estas son de tipo GET) a la ruta `localhost:3000/colors` y deberíamos obtener un json con todos los colores.

P.D.: es posible que el servidor no arranque ya que todavía no hemos creado los routing de usuario ni de de paletas, si ocurre esto comentad las siguientes lineas en `server.js`.

```

//app.use("/users", users);
//app.use("/palettes", palettes);

```

👏👏👏 ¡¡Hemos creado la funcionalidad de colores!!

Como mencionamos antes tenemos en "stand by" a las paletas y a los usuarios. Vamos a continuar con la autenticación y todo lo relacionado con los usuarios, una vez finalizado lo relacionado con los usuarios vamos a terminar con la configuración de `palette.routes.js` y `palette.controller.js`.

Autenticación con JWT

Para securizar las rutas mediante JWT vamos a empezar por definir el `user.controller.js`, este controlador tendrá las funciones de crear un usuario, logout y la mas interesante que es la de autenticar, en la cual se generará un token con el que más tarde con ayuda de un middleware iremos verificando para permitir o no usar ciertas rutas.

```

// Cargamos el modelo
const User = require("../models/User");
// Cargamos el módulo de bcrypt
const bcrypt = require("bcrypt");
// Cargamos el módulo de jsonwebtoken
const jwt = require("jsonwebtoken");
// Cargamos el fichero de los HTTPSTATUSCODE
const HTTPSTATUSCODE = require("../utils/httpStatusCode");

// Codificamos las operaciones que se podran realizar con relacion a los usuarios
const createUser = async (req, res, next) => {
  try {
    const newUser = new User();
    newUser.name = req.body.name;
    newUser.emoji = req.body.emoji;
    newUser.email = req.body.email;
    newUser.password = req.body.password;
    newUser.favPalettes = [];

    //Pnt. mejora: comprobar si el user existe antes de guardar

    const userDb = await newUser.save();

    //Pnt. mejora: autenticar directamente al usuario

    return res.json({
      status: 201,
      message: HTTPSTATUSCODE[201],
      data: null
    });
  }
};

```

```

    } catch (err) {
      return next(err);
    }
  }

const authenticate = async (req, res, next) => {
  try {
    //Buscamos al user en bd
    const userInfo = await User.findOne({ email: req.body.email })
    //Comparamos la contraseña
    if (bcrypt.compareSync(req.body.password, userInfo.password)) {
      //eliminamos la contraseña del usuario
      userInfo.password = null
      //creamos el token con el id y el name del user
      const token = jwt.sign(
        {
          id: userInfo._id,
          name: userInfo.name
        },
        req.app.get("secretKey"),
        { expiresIn: "1h" }
      );
      //devolvemos el usuario y el token.
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { user: userInfo, token: token },
      });
    } else {
      return res.json({ status: 400, message: HTTPSTATUSCODE[400], data: null });
    }
  } catch (err) {
    return next(err);
  }
}

//funcion logout, iguala el token a null.
const logout = (req, res, next) => {
  try {
    return res.json({
      status: 200,
      message: HTTPSTATUSCODE[200],
      token: null
    });
  } catch (err) {
    return next(err)
  }
}

module.exports = {
  createUser,
  authenticate,
  logout
}

```

Para que funcione JWT debemos añadir en el archivo `server.js` lo siguiente:

```

//Config app
app.set("secretKey", "nodeRestApi"); // jwt secret token

```

Ya tenemos creado nuestro controlador de usuarios ahora vamos a configurar el archivo `user.routes.js` que se encargara de gestionar las rutas como ya hemos mencionado con anterioridad.

```

const express = require("express");
const router = express.Router();
//importamos las funciones del controlador y del middleware
const { createUser, authenticate, logout } = require("../controllers/user.controller");
const { isAuthenticated } = require("../middlewares/auth.middleware")

router.post("/register", createUser);
router.post("/authenticate", authenticate);
//le añadimos el middleware para que solo sea accesible si el user esta logueado

```

```
router.post("/logout", [isAuth], logout)

module.exports = router;
```

Este archivo es similar al de los colores, pero como podrás observar estamos importando isAuth. Este va a ser nuestro middleware que se va a encargar de verificar el token y añadir la info del usuario en la petición.

Cuando lo tengamos configurado, si un usuario que no esta logueado se quiere desloguear el middleware se va a encargar de rechazarle y no va a permitir que se llame a la función logout del controlador.

Middlewares

Ahora vamos a trabajar en el archivo `auth.middleware.js`. Este middleware se va a encargar de verificar que la petición trae un token válido y dar permiso para continuar o arrojar un error ya que el token no es válido.

```
//importamos jwt
const jwt = require("jsonwebtoken")

const isAuth = (req, res, next) => {
  //guardamos en una variable la información de la autorizacion de la cabecera
  //de la petición
  const authorization = req.headers.authorization;
  //si no existe autorizacion, no hay token y se retorna error
  if(!authorization){
    return res.json({
      status: 401,
      message: HTTPSTATUSCODE[401],
      data: null
    })
  }
  //si hay token, lo "troceamos " para separar la parte bearer de la info
  //del token
  const splits = authorization.split(" ")
  if( splits.length!=2 || splits[0]!="Bearer"){
    return res.json({
      status: 400,
      message: HTTPSTATUSCODE[400],
      data: null
    })
  }
  //guardamos la info del token en una variable
  const jwtString = splits[1];
  try{
    //verificamos el token y si es ok lo guardamos en una variable
    var token = jwt.verify(jwtString, req.app.get("secretKey"));

    //console.log("token tras verify",token)
  } catch(err){
    return next(err)
  }
  //creamos una variable con la informacion que queremos meter en la
  //petición
  const authority = {
    id : token.id,
    name: token.name
  }
  //Se la asignamos al request de la petición
  req.authority = authority
  //si todo ha ido bien pasamos el middleware
  next()
}
//exportamos la funcion isAuth
module.exports = {
  isAuth,
}
```

Ya tenemos terminada la parte de usuarios de nuestra API. Solo nos queda configurar todo lo relacionado con las paletas y habremos terminado.

Terminando la configuración de nuestra API

Acabamos de entrar en el punto final de la construcción de nuestra API, nos queda terminar la parte de las paletas, para ello vamos a empezar por configurar el archivo `palette.routes.js` y posteriormente vamos a terminar con el archivo `palette.controller.js`.

`palette.routes.js`

```
//importamos Router de express
const express = require("express");
const router = express.Router();
//importamos nuestro middleware
const { isAuth } = require("../middlewares/auth.middleware");
//importamos las funciones del controlador
const {
  newPalette,
  getAllPalettes,
  getPalettesById,
  deletePaletteById,
  updatePaletteById,
  getAllPalettesByUser
} = require("../controllers/palette.controller");

//ruta para crear paleta
router.post("/create", [isAuth], newPalette);
//ruta para obtener todas las paletas
router.get("/", [isAuth], getAllPalettes);
//ruta para obtener las paletas de un usuario
router.get("/palettesbyuser", [isAuth], getAllPalettesByUser);
//ruta para obtener una paleta por id
router.get("/:paletteId", [isAuth], getPalettesById);
//ruta para borrar una paleta
router.delete("/:paletteId", [isAuth], deletePaletteById);
//ruta para modificar una paleta
router.put("/:paletteId", [isAuth], updatePaletteById);
//exportamos las rutas
module.exports = router;
```

Como puedes observar las paletas va a ser el modelo que va a tener mas rutas y funciones, es lógico ya que la API trata de paletas 🍷, las cuales sólo se van a poder consumir cuando el usuario este logueado.

Vamos a darle forma al archivo `palette.controller.js`.

```
// Cargamos el modelo recién creado
const Palette = require("../models/Palette");
// Cargamos el fichero de los HTTPSTATUSCODE
const HTTPSTATUSCODE = require("../utils/httpStatusCode");

//Codificamos las operaciones que se podran realizar con relacion a los paletas

// Metodo para crear una nueva paleta
const newPalette = async (req, res, next) => {
  try {
    //console.log("req.authority", req.authority)
    const newPalette = new Palette();
    newPalette.name = req.body.name;
    newPalette.description = req.body.description;
    newPalette.colors = req.body.colors;
    newPalette.author = req.authority.id; ///este id usuario lo sacamos del token/user logueado
    const paletteDb = await newPalette.save()
    return res.json({
      status: 201,
      message: HTTPSTATUSCODE[201],
      data: { palettes: paletteDb }
    })
  } catch (error) {
    next(error);
  }
}
```

```

    });
  } catch (err) {
    return next(err);
  }
}

//Metodo para retornar todas las paletas registradas en la base de datos
const getAllPalettes = async (req, res, next) => {
  try {
    if (req.query.page) {
      const page = parseInt(req.query.page);
      const skip = (page - 1) * 20;
      const palettes = await Palette.find().skip(skip).limit(20).populate("colors");
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { palettes: palettes },
      });
    } else {
      const palettes = await Palette.find().populate("colors");
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { palettes: palettes },
      });
    }
  } catch (err) {
    return next(err);
  }
};

// Metodo para la busqueda de paletas por ID
const getPalettesById = async (req, res, next) => {
  try {
    const { paletteId } = req.params;
    const paletteDb = await Palette.findById(paletteId).populate("colors");
    return res.json({
      status: 200,
      message: HTTPSTATUSCODE[200],
      data: { palettes: paletteDb },
    });
  } catch (err) {
    return next(err);
  }
};

//Metodo para eliminar algun registro de la base de datos
const deletePaletteById = async (req, res, next) => {
  try {
    const { paletteId } = req.params;
    const authority = req.authority.id
    const userPalette = await Palette.findById(paletteId)

    if (authority == userPalette.author._id) {

      const paletteDeleted = await Palette.findByIdAndDelete(paletteId);
      if (!paletteDeleted) {
        return res.json({
          status: 200,
          message: "There is not a palette with that Id",
          data: null
        });
      } else {
        return res.json({
          status: 200,
          message: HTTPSTATUSCODE[200],
          data: { palettes: paletteDeleted },
        });
      }
    } else {
      return res.json({
        status: 403,
        message: HTTPSTATUSCODE[403],
        data: null
      });
    }
  } catch (err) {
    return next(err);
  }
};

```



```
//Metodo para actualizar algun registro de la base de datos
const updatePaletteById = async (req, res, next) => {
  try {
    const { paletteId } = req.params;
    const authority = req.authority.id
    const userPalette = await Palette.findById(paletteId)

    if (authority == userPalette.author._id) {

      const paletteToUpdate = new Palette();
      if (req.body.name) paletteToUpdate.name = req.body.name;
      if (req.body.description) paletteToUpdate.description = req.body.description;
      if (req.body.colors) paletteToUpdate.colors = req.body.colors;
      paletteToUpdate._id = paletteId;

      const paletteUpdated = await Palette.findByIdAndUpdate(paletteId, paletteToUpdate);
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { palettes: paletteUpdated }
      });
    } else {
      return res.json({
        status: 403,
        message: HTTPSTATUSCODE[403],
        data: null
      })
    }
  } catch (err) {
    return next(err);
  }
}

//Metodo para obtener los documentos de la bd segun el usuario que lo ha creado
const getAllPalettesByUser = async (req, res, next) => {
  try {
    const author = req.authority.id;

    if (req.query.page) {
      const page = parseInt(req.query.page);
      const skip = (page - 1) * 20;
      const allPalettesByUser = await Palette.find({ author: author }).skip(skip).limit(20).populate("colors");
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { palettes: allPalettesByUser },
      });
    } else {
      const allPalettesByUser = await Palette.find({ author: author }).populate("colors");
      return res.json({
        status: 200,
        message: HTTPSTATUSCODE[200],
        data: { palettes: allPalettesByUser },
      });
    }
  } catch (err) {
    return next(err)
  }
}

//Exportamos la funciones
module.exports = {
  newPalette,
  getAllPalettes,
  getPalettesById,
  deletePaletteById,
  updatePaletteById,
  getAllPalettesByUser
}
```

🎉🎉🎉 ¡¡¡Terminamos nuestra API !!! 🎉🎉🎉

Espero que esta guía te haya servido y te sea de utilidad. Ahora sólo nos queda animarte a que implementes todas las mejoras que veas oportunas. 🐦

