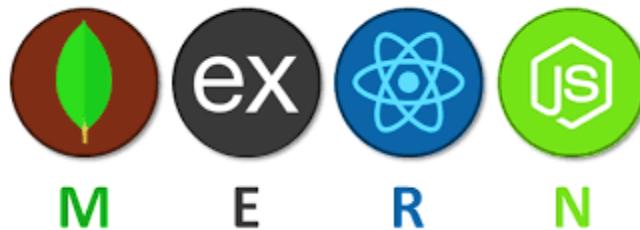


Full Stack MERN



¿Qué es MERN?

MERN es un acrónimo que se utiliza para describir un conjunto de tecnologías de desarrollo web utilizadas juntas para construir aplicaciones web modernas y escalables. Las letras en MERN representan las siguientes tecnologías:

- [MongoDB](#): una base de datos NoSQL que utiliza JSON como formato de almacenamiento de datos.
- [Express](#): un framework de aplicación web de Node.js que proporciona una estructura sólida para construir aplicaciones web.
- [React](#): una biblioteca de JavaScript de código abierto utilizada para construir interfaces de usuario interactivas y escalables.
- [Node.js](#): un entorno de tiempo de ejecución de JavaScript que permite ejecutar código de JavaScript en el servidor.

En conjunto, MERN proporciona una pila de tecnología de desarrollo web completa y escalable para construir aplicaciones web modernas y altamente eficientes.

MERN se utiliza en una amplia variedad de aplicaciones web modernas y escalables. Algunos ejemplos de aplicaciones que utilizan MERN incluyen:

- Redes sociales: aplicaciones web como Facebook, Twitter y LinkedIn utilizan MERN para construir interfaces de usuario interactivas y escalables, y almacenar y administrar grandes cantidades de datos de usuarios.
- Aplicaciones de comercio electrónico: sitios web como Amazon, eBay y Alibaba utilizan MERN para construir plataformas de compras en línea escalables y eficientes.
- Aplicaciones de medios: plataformas de transmisión de video como Netflix y YouTube utilizan MERN para construir interfaces de usuario escalables y manejar grandes cantidades de datos de contenido.
- Aplicaciones de productividad: aplicaciones web como Trello y Slack utilizan MERN para construir interfaces de usuario interactivas y escalables para la gestión de proyectos y la colaboración en equipo.



@hdtoledo

Requisitos para iniciar con el curso

- Nociones básicas de JavaScript
- Nociones básicas de React JS

Descripción

En este curso vas a aprender a crear una aplicación web que será una web personal con panel de Administrador protegido por un login con **JWT**, todo paso a paso usando el **MERN Stack** que está compuesto por **MongoDB**, **Express JS**, **React JS** y **Node JS**.

Crearemos nuestra aplicación desde cero sin usar nada prefabricado, **aprenderemos base de datos no relacional con MongoDB**, en el **Backend crearemos un API REST** con **Node JS** y **Express JS** y en el **Frontend usaremos React JS** con **Hooks** y en la **parte del CSS usaremos SASS**.

Cuando tengamos nuestra aplicación terminada, aprenderemos a **desplegar nuestra aplicación en la nube**.

Este curso tiene como objetivo enseñarte a desarrollar cualquier tipo de aplicación desde cero, **convirtiéndote en un desarrollador Full Stack** sobre el **MERN Stack**.

Estructura del curso

- **Instalación y configuración** del entorno de trabajo.
- **Desacoplaremos** nuestro **proyecto** en **tres bloques**, Base de Datos, Backend y Frontend.
- **Crearemos una API REST** desde cero conectada a MongoDB.
- Aprenderemos a usar el **ODM Mongoose**.
- **Añadiremos** al Frontend **SASS**.
- **Crearemos** una **configuración dinámica** de **React Router Dom**.
- Crearemos Sistemas de Layouts.
- Creamos un **sistema de Auth protegido con JWT** y tendremos el **AccessToken para permitir acceso** y **RefreshToken para recuperar sesiones**.
- **Creamos un panel de Administrador** para que nuestros usuarios con privilegios puedan **gestionar la web**.
- Construiremos un Menú completamente dinámico gestionado desde el panel de **administrador**.
- Crearemos una **Home Page llamativa**.
- Programaremos un **Sistema de Newsletter** 100% funcional **desde cero**.
- Crearemos una **sección para subir cursos** conectada a una **API**.

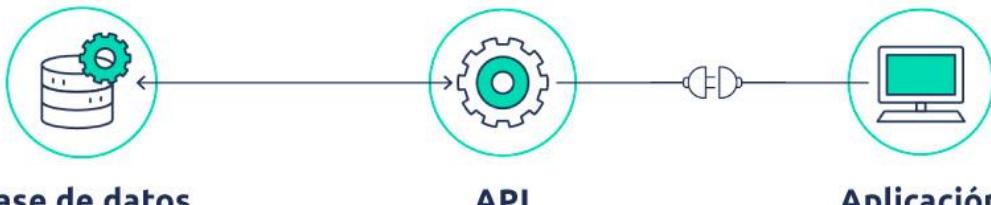


@hdtoledo

- Crearemos un **Sistema de Blog**, con **paginación y creación de URL dinámicas** todo **gestionado desde el panel de Administrador**.
- Gestionaremos el **SEO On Page** para mejorar nuestra visibilidad en **Google**.
- Desplegaremos nuestra aplicación en **varios servidores en la nube**.

Cliente API

¿Cómo funciona una API?



Base de datos

La información o las herramientas desarrolladas por una empresa son utilizadas en servicios de terceros.

API

Una API permite conectar la información o funcionalidades con los requerimientos de una aplicación.

Aplicación

El cliente tiene acceso a toda su información requerida en una sola aplicación.

Un cliente API es una aplicación o componente de software que se utiliza para interactuar con una API (Interfaz de Programación de Aplicaciones).

Una API es un conjunto de reglas y protocolos que permite a diferentes aplicaciones comunicarse y compartir datos entre sí. El cliente API actúa como un intermediario entre el usuario o la aplicación y la API, enviando solicitudes a la API y recibiendo respuestas en un formato específico, como JSON o XML.

El cliente API puede ser desarrollado en diferentes lenguajes de programación y se utiliza para realizar operaciones como enviar solicitudes HTTP, autenticarse con la API, enviar parámetros y recibir datos de respuesta. Por ejemplo, un cliente API puede utilizarse para acceder a servicios web, obtener información de una base de datos remota o interactuar con plataformas de redes sociales.

En resumen, un cliente API es un componente de software que permite a una aplicación o usuario interactuar con una API para acceder y utilizar los servicios y datos proporcionados por la API.

Algunas de ellas:

1. [Insomnia](#) es una herramienta de cliente API de código abierto y multiplataforma que se utiliza para realizar solicitudes HTTP y probar APIs. Proporciona una interfaz gráfica intuitiva que permite a los desarrolladores enviar solicitudes a una API, inspeccionar las respuestas y depurar problemas de comunicación.

Insomnia ofrece características útiles como autocompletado de URL y parámetros, resultado de sintaxis, guardado de solicitudes y respuestas, gestión de cookies, autenticación y soporte para varios tipos de autenticación (como API key, token de acceso, OAuth, etc.), entre otras.

La herramienta es especialmente útil durante el desarrollo y la integración de APIs, ya que permite realizar pruebas exhaustivas, guardar configuraciones y compartir solicitudes entre miembros del equipo. Además, Insomnia admite la importación y exportación de colecciones de solicitudes, lo que facilita la colaboración y el intercambio de configuraciones de API.

En resumen, Insomnia es una herramienta poderosa y versátil para interactuar con APIs y realizar pruebas de API de manera eficiente.

2. **Postman:** Una herramienta de colaboración y desarrollo de API que permite enviar solicitudes y recibir respuestas de APIs. Proporciona una interfaz gráfica fácil de usar para probar y depurar APIs.
3. **cURL:** Una herramienta de línea de comandos que permite enviar y recibir datos utilizando varios protocolos, incluyendo HTTP, FTP y SMTP. Es muy utilizado para interactuar con APIs mediante solicitudes HTTP.
4. **Axios:** Una biblioteca de JavaScript ampliamente utilizada para realizar solicitudes HTTP desde aplicaciones web o móviles. Proporciona una interfaz fácil de usar y admite promesas para un código más legible y conciso.
5. **Retrofit:** Una biblioteca de Android que simplifica la comunicación con servicios web a través de API RESTful. Proporciona una forma sencilla de definir y realizar solicitudes HTTP en aplicaciones Android.

[Node.js](#)

Node.js es un entorno de ejecución de código abierto basado en el motor de JavaScript V8 de Chrome. A diferencia de la ejecución de JavaScript en el navegador, Node.js permite ejecutar JavaScript en el lado del servidor, lo que lo convierte en una opción popular para el desarrollo de aplicaciones web y servidores.

Node.js utiliza un enfoque asíncrono y basado en eventos, lo que significa que puede manejar múltiples solicitudes simultáneamente sin bloquear el flujo de ejecución. Esto se logra utilizando el bucle de eventos y el modelo de E/S no bloqueante, lo que permite una escalabilidad eficiente y un rendimiento de alta concurrencia.

Algunas características clave de Node.js son:

1. **JavaScript en el lado del servidor:** Permite a los desarrolladores utilizar el mismo lenguaje de programación tanto en el lado del cliente como en el lado del servidor, lo que facilita el intercambio de código y la reutilización de habilidades.
2. **Eficiencia y rendimiento:** Node.js utiliza un enfoque sin bloqueo y basado en eventos, lo que le permite manejar grandes cargas de trabajo con eficiencia y alta concurrencia.



3. Amplio ecosistema de módulos: Node.js cuenta con un gestor de paquetes llamado npm (Node Package Manager), que permite a los desarrolladores acceder a un vasto repositorio de módulos y paquetes preexistentes para facilitar el desarrollo de aplicaciones.
4. Desarrollo rápido de prototipos: La facilidad de uso y la naturaleza flexible de JavaScript, junto con la disponibilidad de numerosos módulos de terceros, hacen que Node.js sea una opción popular para el desarrollo rápido de prototipos y la construcción de aplicaciones escalables.

Node.js se utiliza en una amplia gama de aplicaciones, desde el desarrollo de servidores web y aplicaciones de tiempo real hasta herramientas de línea de comandos y aplicaciones de Internet de las cosas (IoT).

Gestor de paquetes

Un gestor de paquetes es una herramienta que facilita la instalación, gestión y actualización de paquetes de software en un entorno de desarrollo o producción. Los paquetes son componentes de software preempaquetados que contienen código, bibliotecas, dependencias y otros recursos necesarios para realizar una funcionalidad específica.

Los gestores de paquetes simplifican el proceso de adquisición y administración de paquetes, proporcionando una forma estandarizada y automatizada de manejar las dependencias y versiones de software. Algunos ejemplos populares de gestores de paquetes son npm (Node Package Manager) para Node.js, pip para Python, gem para Ruby y apt-get para sistemas basados en Debian/Ubuntu.

Las principales funciones de un gestor de paquetes incluyen:

1. Instalación: Permite descargar e instalar paquetes de software en un entorno determinado. El gestor de paquetes se encarga de manejar las dependencias y asegurarse de que todas las dependencias necesarias también se instalen correctamente.
2. Gestión de dependencias: Maneja las dependencias entre los paquetes, asegurándose de que todas las dependencias requeridas estén presentes y en las versiones correctas. Esto simplifica la configuración del entorno de desarrollo y garantiza que las aplicaciones funcionen correctamente.
3. Actualización: Permite actualizar los paquetes instalados a nuevas versiones disponibles. Los gestores de paquetes pueden verificar y descargar automáticamente las actualizaciones, facilitando la tarea de mantener el software actualizado.
4. Control de versiones: Administra las diferentes versiones de los paquetes, permitiendo instalar versiones específicas o manejar múltiples versiones simultáneamente. Esto es especialmente útil cuando se trabaja en proyectos que requieren versiones específicas de bibliotecas o dependencias.

En resumen, un gestor de paquetes es una herramienta esencial para simplificar la gestión y distribución de software, asegurando la consistencia y facilitando la colaboración en proyectos de desarrollo de software.

Yarn

Yarn es un gestor de paquetes de código abierto desarrollado por Facebook que se utiliza principalmente en proyectos de JavaScript. Al igual que npm (Node Package Manager), Yarn permite instalar, gestionar y actualizar paquetes de software y sus dependencias.

Sin embargo, Yarn se diseñó con el objetivo de mejorar la velocidad, la eficiencia y la confiabilidad en comparación con npm. Algunas características destacadas de Yarn son:

1. Resolución de dependencias determinista: Yarn utiliza un algoritmo de resolución de dependencias más robusto que garantiza que las mismas dependencias se instalen en todas las máquinas, evitando problemas de inconsistencias entre los entornos de desarrollo.
2. Instalaciones paralelas: Yarn realiza las instalaciones de paquetes de manera paralela, aprovechando al máximo la capacidad del hardware y acelerando el proceso de instalación.
3. Cache de paquetes: Yarn utiliza un sistema de caché local que almacena las versiones descargadas de los paquetes, lo que permite una instalación más rápida y eficiente en futuros proyectos o actualizaciones.
4. Resolución de conflictos automática: En caso de conflictos en las versiones de las dependencias, Yarn resuelve automáticamente los conflictos y selecciona la mejor versión disponible según las reglas de resolución establecidas.
5. Integridad de los paquetes: Yarn verifica la integridad de los paquetes instalados mediante el uso de sumas de verificación (checksums), asegurando que los paquetes descargados no hayan sido modificados o corrompidos.

En resumen, Yarn es un gestor de paquetes que proporciona mejoras en la velocidad, eficiencia y confiabilidad en comparación con npm. Es especialmente útil en proyectos de JavaScript con muchas dependencias, ya que ayuda a garantizar la consistencia y la estabilidad del entorno de desarrollo.

MongoDB

The screenshot shows the MongoDB homepage on the left and a terminal window on the right. The homepage features the MongoDB logo, navigation links for Products, Solutions, Resources, Company, and Pricing, and a 'Try Free' button. The main heading is 'Build the next big thing'. Below it is a subtext: 'The developer data platform that provides the services and tools necessary to build distributed applications fast, at the performance and scale users demand.' It also includes 'Start Free' and 'Documentation' buttons. The terminal window on the right displays a command-line interface connecting to a MongoDB cluster: 'Connecting to: mongodb+srv://cluster0.ab123.mongodb.net/?retryWrites=true&w=majority' and 'Using MongoDB: 6.0'. The output lists various MongoDB regions and their names: South America - southamerica-west1, South America - brazil-southeast, Northern America - northamerica-northeast2, South America - northamerica-northeast1, Asia Pacific - asia-south2, Australia - australia-southeast2, Europe - europe-central2, Middle East - europe-central2, Africa - europe-central2, Asia Pacific - ap-northeast-3, North America - us-gov-west-1, Northern America - us-gov-east-1.



MongoDB es un sistema de base de datos NoSQL (no relacional) que se ha vuelto popular debido a su flexibilidad y escalabilidad. Fue diseñado para manejar grandes volúmenes de datos de manera eficiente y brindar un acceso rápido a la información almacenada.

A diferencia de las bases de datos relacionales tradicionales, como MySQL o PostgreSQL, que utilizan tablas y esquemas predefinidos, MongoDB se basa en un modelo de documentos. En lugar de almacenar los datos en filas y columnas, MongoDB almacena los datos en documentos JSON (JavaScript Object Notation), que son estructuras de datos flexibles y autocontenidoas.

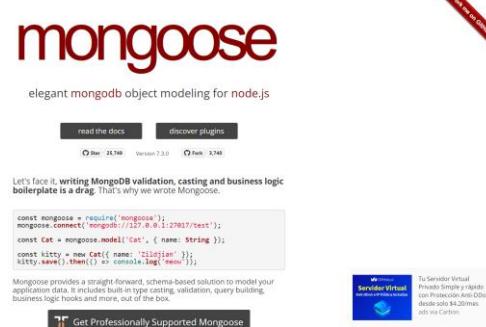
Cada documento en MongoDB se compone de pares clave-valor, donde la clave es el nombre del campo y el valor puede ser de varios tipos, como cadenas de texto, números, arreglos, objetos incrustados, etc. Esto permite una representación muy flexible de los datos y la capacidad de almacenar información relacionada dentro de un solo documento.

MongoDB también proporciona una potente consulta y lenguaje de manipulación de datos llamado MongoDB Query Language (MQL). Con MQL, puedes realizar consultas sofisticadas para buscar, filtrar, actualizar y eliminar datos en la base de datos.

Además, MongoDB está diseñado para ser escalable horizontalmente, lo que significa que puedes distribuir la carga de trabajo en varios servidores o clústeres. Esto permite manejar grandes volúmenes de datos y un alto rendimiento en entornos de aplicaciones con alta demanda.

En resumen, MongoDB es una base de datos NoSQL que utiliza un modelo de documentos flexible y escalable. Es especialmente útil para aplicaciones que manejan grandes cantidades de datos y requieren una alta flexibilidad en el esquema de datos.

[Mongoose](#)



Mongoose es una biblioteca de modelado de objetos para Node.js que proporciona una interfaz basada en esquemas para interactuar con bases de datos MongoDB. En pocas palabras, Mongoose es una capa de abstracción que simplifica y facilita el trabajo con MongoDB en aplicaciones Node.js.

Mongoose se utiliza comúnmente junto con MongoDB para definir la estructura de los datos, validarlos y proporcionar métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos de manera más intuitiva.

Algunas de las características principales de Mongoose son las siguientes:



1. Modelado de datos: Mongoose permite definir modelos con esquemas claros y consistentes. Los esquemas describen la estructura de los datos, incluidos los campos, los tipos de datos permitidos, las validaciones, los índices y otras configuraciones.
2. Validación de datos: Puedes utilizar los esquemas de Mongoose para aplicar reglas de validación a los datos antes de ser guardados en la base de datos. Esto asegura que los datos cumplan con los requisitos establecidos, como campos requeridos, tipos de datos específicos, longitud de cadenas, entre otros.
3. Middleware: Mongoose también permite agregar funciones de middleware para ejecutar código antes o después de ciertas operaciones de base de datos, como guardar, actualizar o eliminar un documento. Esto es útil para realizar tareas adicionales, como encriptar contraseñas, generar valores predeterminados, entre otros.
4. Consultas y agregaciones: Mongoose proporciona una API rica para realizar consultas y agregaciones en la base de datos. Puedes utilizar métodos encadenados y operadores para buscar datos, ordenarlos, filtrarlos y realizar cálculos de agregación más complejos.
5. Integración con Express y Node.js: Mongoose se integra bien con el marco de desarrollo web Express.js y se utiliza comúnmente en aplicaciones Node.js para interactuar con la base de datos MongoDB de manera sencilla y eficiente.

En resumen, Mongoose es una biblioteca de modelado de objetos para Node.js que simplifica la interacción con MongoDB al proporcionar una interfaz basada en esquemas. Facilita la definición de modelos de datos, la validación, las consultas y otras operaciones comunes en la base de datos.

IDE (Entornos de Desarrollo Integrado)

Hay varios IDE (Entornos de Desarrollo Integrado) populares que puedes considerar según el lenguaje de programación que deseas utilizar. Aquí hay algunas opciones recomendadas para diferentes lenguajes:

1. [Visual Studio Code \(VS Code\)](#): Es un editor de código gratuito y altamente personalizable desarrollado por Microsoft. Es ampliamente utilizado y admite una amplia gama de lenguajes de programación. Tiene una gran cantidad de extensiones disponibles que te permiten personalizar y ampliar sus capacidades según tus necesidades.
2. PyCharm: Es un IDE de JetBrains diseñado específicamente para el desarrollo de Python. Proporciona características avanzadas para la edición, depuración y análisis de código Python. Hay una versión gratuita llamada PyCharm Community Edition, así como una versión de pago llamada PyCharm Professional con características adicionales.
3. IntelliJ IDEA: También desarrollado por JetBrains, IntelliJ IDEA es un IDE potente y versátil que admite varios lenguajes de programación, incluidos Java, Kotlin, Scala, JavaScript y más. Además de sus características estándar, ofrece herramientas inteligentes de refactorización, análisis de código y soporte para frameworks populares.
4. Eclipse: Es un IDE gratuito y de código abierto que es ampliamente utilizado para el desarrollo en Java, pero también admite otros lenguajes a través de complementos. Eclipse tiene una gran



@hdtoledo

comunidad y una amplia gama de complementos disponibles que lo hacen altamente personalizable y adecuado para proyectos de diferentes tamaños.

5. Xcode: Si te interesa el desarrollo de aplicaciones para macOS, iOS, watchOS o tvOS, Xcode es el IDE oficial de Apple. Proporciona un conjunto completo de herramientas para el desarrollo de aplicaciones en Swift y Objective-C, así como para la creación de interfaces de usuario y la depuración en dispositivos de Apple.

Estas son solo algunas opciones populares, pero hay muchos otros IDE disponibles según tus necesidades y preferencias. Considera los lenguajes de programación que deseas utilizar, las características que necesitas y la comunidad y soporte disponibles al elegir un IDE para programar.

Extensiones para tener en cuenta en VS code

The screenshot displays five recommended extensions for Visual Studio Code:

- Bracket Pair Colorization Toggler** v0.0.3 by Dzhavat Ushev (610.204 installs, 4 stars): Quickly toggle 'Bracket Pair Colorization' setting with a simple command. Status: Enabled globally.
- ES7 React/Redux/GraphQL/React-Native snippets** v1.9.3 by rodrigovalladas (711.209 installs, 4 stars): Simple extensions for React, Redux and GraphQL in JS/TS with ES7 syntax (forked from dsznajder). Status: Enabled globally.
- ESLint** v2.4.0 by Microsoft (microsoft.com) (27.640.926 installs, 5 stars): Integrates ESLint JavaScript into VS Code. Status: Enabled globally.
- IntelliSense for CSS class names in HTML** v1.20.0 by Zignld (6.293.804 installs, 5 stars): CSS class name completion for the HTML class attribute based on the definitions found in your workspace. Status: Enabled globally.
- JavaScript (ES6) code snippets** v1.8.0 by charalampos karypidis (12.342.838 installs, 5 stars): Code snippets for JavaScript in ES6 syntax. Status: Enabled globally.





Hablemos de [React](#)

[wiki](#)

React es una biblioteca de JavaScript de código abierto para construir interfaces de usuario. Está basada en la componentización de la UI: la interfaz se divide en componentes independientes, que contienen su propio estado. Cuando el estado de un componente cambia, React vuelve a renderizar la interfaz.

Esto hace que React sea una herramienta muy útil para construir interfaces complejas, ya que permite dividir la interfaz en piezas más pequeñas y reutilizables.

Fue creada en 2011 por Jordan Walke, un ingeniero de software que trabajaba en Facebook y que quería simplificar la forma de crear interfaces de usuario complejas. Es una biblioteca muy popular y es usada por muchas empresas como Facebook, Netflix, Airbnb, Twitter, Instagram, etc.

```
state = {
  products: storeProducts
}

render() {
  return (
    <React.Fragment>
      <div className="py-5">
        <div className="container">
          <Title name="our" title="products" />
          <div className="row">
            <ProductConsumer>
              {(value) => {
                console.log(value)
              }}
            </ProductConsumer>
          </div>
        </div>
      </React.Fragment>
    )
}
```

¿Qué es JSX?

JSX (JavaScript XML) es una extensión de sintaxis utilizada en React, una biblioteca de JavaScript para construir interfaces de usuario. JSX permite combinar HTML y JavaScript en un solo archivo, lo que facilita la creación de componentes reutilizables y dinámicos.



@hdtledo

En JSX, se pueden escribir elementos y componentes de React utilizando una sintaxis similar a HTML, pero también se pueden incluir expresiones de JavaScript dentro de las etiquetas utilizando llaves {}. Esto permite la manipulación dinámica de datos y la generación de contenido basado en lógica.

```
import React from 'react';

// Definición de un componente de React utilizando JSX
const Saludo = ({ nombre }) => {
  return <h1>Hola, {nombre}!</h1>;
};

// Utilización del componente en otra parte de la aplicación
const App = () => {
  const usuario = 'John Doe';
  return (
    <div>
      <Saludo nombre={usuario} />
    </div>
  );
};

export default App;
```

En el ejemplo anterior, se define un componente de React llamado **Saludo** que recibe una prop **nombre**. En el componente principal **App**, se utiliza el componente **Saludo** pasando el valor '**John Doe**' como prop **nombre**. El componente **Saludo** renderiza un elemento **h1** que muestra el saludo personalizado.

Gracias a JSX, es posible combinar de manera elegante y legible HTML y JavaScript en un solo archivo, lo que facilita el desarrollo de interfaces de usuario dinámicas y componentes reutilizables en React.

¿Cuáles son las características principales de React?

Las características principales de React son:

- **Componentes:** React está basado en la componetización de la UI. La interfaz se divide en componentes independientes, que contienen su propio estado. Cuando el estado de un componente cambia, React vuelve a renderizar la interfaz.
- **Virtual DOM:** React usa un DOM virtual para renderizar los componentes. El DOM virtual es una representación en memoria del DOM real. Cuando el estado de un componente cambia, React vuelve a renderizar la interfaz. En lugar de modificar el DOM real, React modifica el DOM virtual y, a continuación, compara el DOM virtual con el DOM real. De esta forma, React sabe qué cambios se deben aplicar al DOM real.



@hdtoledo

- **Declarativo:** React es declarativo, lo que significa que no se especifica cómo se debe realizar una tarea, sino qué se debe realizar. Esto hace que el código sea más fácil de entender y de mantener.
- **Unidireccional:** React es unidireccional, lo que significa que los datos fluyen en una sola dirección. Los datos fluyen de los componentes padres a los componentes hijos.
- **Universal:** React se puede ejecutar tanto en el cliente como en el servidor. Además, puedes usar React Native para crear aplicaciones nativas para Android e iOS.

¿Qué es un componente?

Un componente es una pieza de código que renderiza una parte de la interfaz. Los componentes pueden ser parametrizados, reutilizados y pueden contener su propio estado.

En React los componentes se crean usando funciones o clases.

¿Cuál es la diferencia entre componente y elemento en React?

Un componente es una función o clase que recibe props y devuelve un elemento. Un elemento es un objeto que representa un nodo del DOM o una instancia de un componente de React.

```
// Elemento que representa un nodo del DOM
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}

// Elemento que representa una instancia de un componente
{
  type: Button,
  props: {
    color: 'blue',
    children: 'OK!'
  }
}
```

¿Qué son las props en React?

Las props son las propiedades de un componente. Son datos que se pasan de un componente a otro. Por ejemplo, si tienes un componente Button que muestra un botón, puedes pasarle una prop text para que el botón muestre ese texto:

```
function Button(props) {
  return <button>{props.text}</button>
}
```

Podríamos entender que el componente Button es un botón genérico, y que la prop text es el texto que se muestra en el botón. Así estamos creando un componente reutilizable.



@hdtoledo

Debe considerarse además que al usar cualquier expresión JavaScript dentro de JSX debe envolverlos con {}, en este caso el objeto props, de otra forma JSX lo considerará como texto plano.

Para usarlo, indicamos el nombre del componente y le pasamos las props que queremos:

```
<Button text="Haz clic aquí" />
<Button text="Seguir a @hdtoledo" />
```

¿Qué son los hooks?

Los Hooks son una API de React que nos permite tener estado, y otras características de React, en los componentes creados con una function.

Esto, antes, no era posible y nos obligaba a crear un componente con class para poder acceder a todas las posibilidades de la librería.

Hooks es gancho y, precisamente, lo que hacen, es que te permiten enganchar tus componentes funcionales a todas las características que ofrece React.

¿Qué hace el hook `useState`?

El hook **useState** es un componente fundamental de React que permite agregar estado a componentes funcionales. Antes de la introducción de los hooks en React, los componentes funcionales no tenían capacidad para mantener estado interno. Sin embargo, con el uso del hook **useState**, ahora es posible declarar y actualizar el estado en componentes funcionales.

La función **useState** es un hook incorporado en React que devuelve un array con dos elementos: la variable de estado actual y una función para actualizar esa variable. La sintaxis básica para utilizar **useState** es la siguiente:

```
const [state, setState] = useState(initialState);
```

Donde:

- **state**: es la variable que representa el estado actual.
- **setState**: es la función utilizada para actualizar el estado.
- **initialState**: es el valor inicial del estado.

Cuando se invoca **useState**, se devuelve un array donde el primer elemento (**state**) es el valor actual del estado y el segundo elemento (**setState**) es una función que permite actualizar ese estado. Al llamar a **setState** con un nuevo valor, React re-renderiza el componente y actualiza el valor del estado.

Aquí tienes un ejemplo simple de cómo utilizar **useState**:



```

import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

```

En el ejemplo anterior, el componente **Counter** tiene un estado llamado **count**, inicializado en 0 mediante **useState(0)**. La función **increment** utiliza **setCount** para actualizar el valor de **count** cada vez que se hace clic en el botón. El nuevo valor se refleja automáticamente en la interfaz de usuario a través de la expresión **{count}** en el componente JSX.

En resumen, el hook **useState** en React permite agregar y actualizar el estado en componentes funcionales, lo que facilita la gestión de datos dinámicos y el re-renderizado de componentes en respuesta a cambios en el estado.

Empecemos creando una app de REACT

Vamos a abrir nuestra terminal y allí vamos a utilizar el siguiente comando: **npm create vite**

```

Microsoft Windows [Versión 10.0.22621.1702]
(c) Microsoft Corporation. Todos los derechos reservados.

D:\DATA\Documents>npm create vite
Need to install the following packages:
  create-vite@4.3.1
Ok to proceed? (y) y|

```

Colocamos **Y** y presionamos enter, a continuación, nos preguntara cual es el nombre del proyecto:

```

Microsoft Windows [Versión 10.0.22621.1702]
(c) Microsoft Corporation. Todos los derechos reservados.

D:\DATA\Documents>npm create vite
Need to install the following packages:
  create-vite@4.3.1
Ok to proceed? (y) y
? Project name: » vite-project|

```

Vamos a colocar **react_basics** y presionamos enter:



```
Microsoft Windows [Versión 10.0.22621.1702]
(c) Microsoft Corporation. Todos los derechos reservados.

D:\DATA\Documents>npm create vite
Need to install the following packages:
  create-vite@4.3.1
Ok to proceed? (y) y
✓ Project name: ... react_basics
? Select a framework: » - Use arrow-keys. Return to submit.
>  Vanilla
  Vue
  React
  Preact
  Lit
  Svelte
  Others
```

Acá nos permite seleccionar el framework que vamos a utilizar en este caso seleccionamos React y presionamos enter:

```
Microsoft Windows [Versión 10.0.22621.1702]
(c) Microsoft Corporation. Todos los derechos reservados.

D:\DATA\Documents>npm create vite
Need to install the following packages:
  create-vite@4.3.1
Ok to proceed? (y) y
✓ Project name: ... react_basics
✓ Select a framework: » React
? Select a variant: » - Use arrow-keys. Return to submit.
>  TypeScript
  TypeScript + SWC
  JavaScript
  JavaScript + SWC
```

Ahora seleccionamos javascript y presionamos enter:

```
Microsoft Windows [Versión 10.0.22621.1702]
(c) Microsoft Corporation. Todos los derechos reservados.

D:\DATA\Documents>npm create vite
Need to install the following packages:
  create-vite@4.3.1
Ok to proceed? (y) y
✓ Project name: ... react_basics
✓ Select a framework: » React
✓ Select a variant: » JavaScript

Scaffolding project in D:\DATA\Documents\react_basics...

Done. Now run:

  cd react_basics
  npm install
  npm run dev

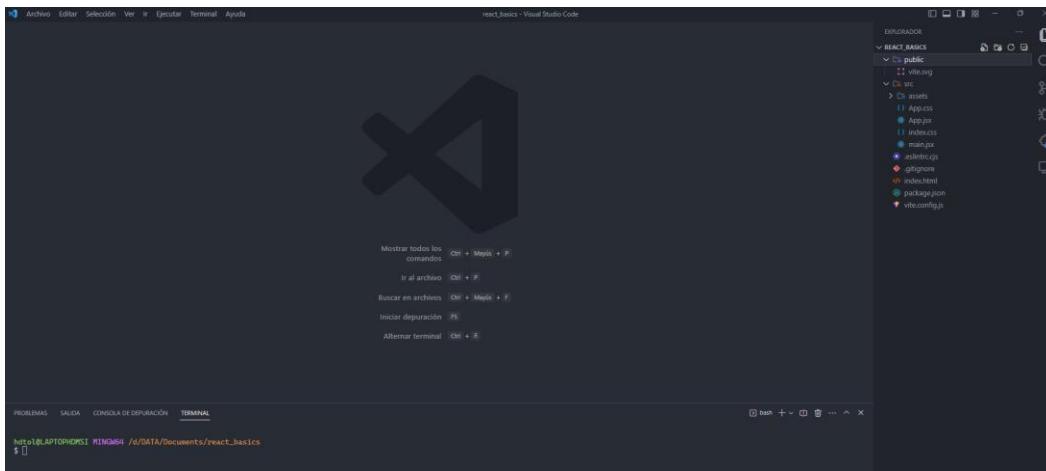
D:\DATA\Documents>
```

Ahora ingresamos a nuestra carpeta recién creada **react_basics** y allí vamos a abrir nuestro VS code con el comando **code .**

```
D:\DATA\Documents\react_basics>code .
```



@hdtoledo



Ahora en nuestro terminal vamos a hacer la instalación de las dependencias a través del comando **npm install**

```
hdtol@LAPTOPHDMI MINGW64 /d/DATA/Documents/react_basics
$ npm install

added 239 packages, and audited 240 packages in 27s

80 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
```

Ahora vamos a ejecutar el comando **npm run dev** para abrir nuestro servidor y poder ver el proyecto en la web:

```
hdtol@LAPTOPHDMI MINGW64 /d/DATA/Documents/react_basics
$ npm run dev

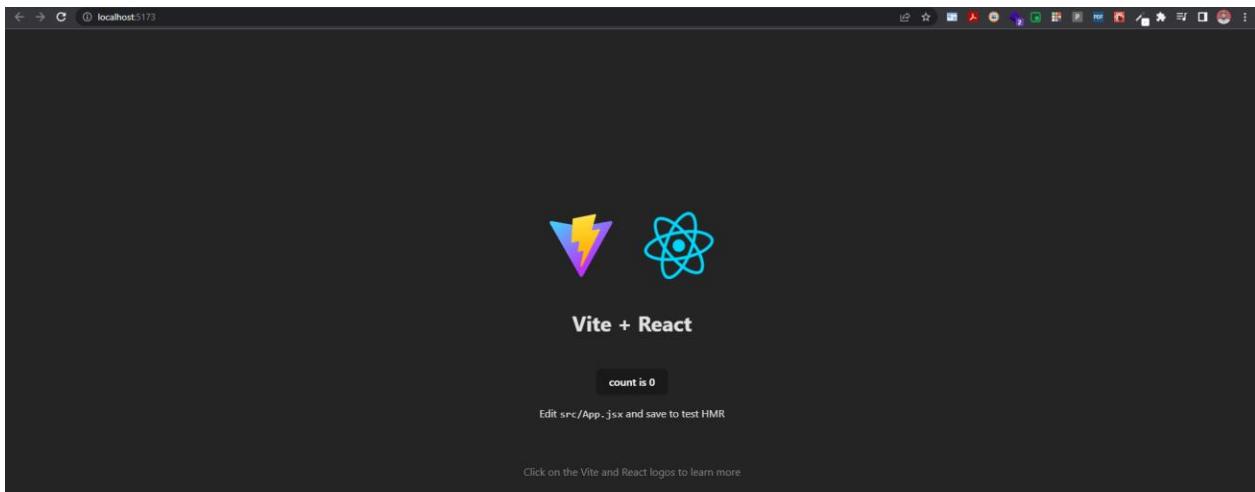
> react_basics@0.0.0 dev
> vite

VITE v4.3.8 ready in 482 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
```

De esta manera ya tendremos nuestro proyecto ejecutándose en el localhost:





Hablemos de [Vite](#) es una herramienta de desarrollo web rápida y liviana creada por Evan You, el creador de Vue.js. Está diseñada para mejorar la experiencia de desarrollo al permitir un tiempo de compilación instantáneo y un servidor de desarrollo extremadamente rápido.

A diferencia de otras herramientas de construcción como webpack o Parcel, que se basan en la creación de un único paquete final antes de ejecutar la aplicación, Vite adopta un enfoque de desarrollo basado en la modularidad y la carga bajo demanda.

En lugar de agrupar todos los archivos de la aplicación en un solo archivo en tiempo de compilación, Vite utiliza la capacidad de los navegadores modernos para importar módulos directamente desde el sistema de archivos local durante el desarrollo. Esto significa que cada módulo individual se compila y se sirve por separado, lo que resulta en un tiempo de compilación y recarga instantáneos cuando se realizan cambios en el código fuente.

Vite también tiene una integración nativa con los marcos de JavaScript más populares, como Vue.js, React y Preact, lo que permite una configuración sencilla y un flujo de desarrollo optimizado para estas tecnologías.

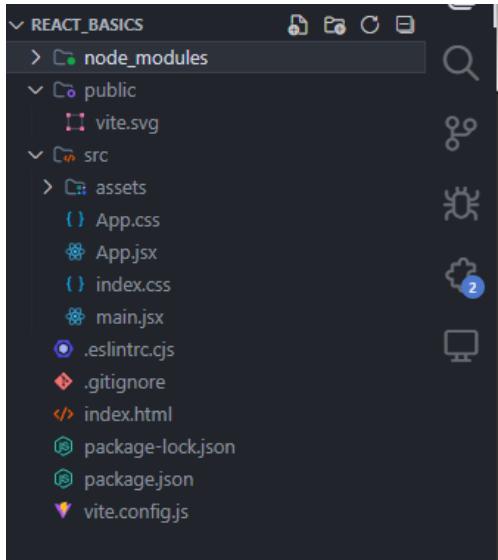
Además de su velocidad de desarrollo, Vite ofrece otras características útiles, como la recarga en caliente (hot module replacement) que actualiza solo los componentes modificados en lugar de recargar toda la página, un entorno de desarrollo con soporte para TypeScript y CSS preprocesados, y la capacidad de generar un paquete optimizado y minificado para la producción.

En resumen, Vite es una herramienta de desarrollo web rápida y eficiente que mejora la experiencia de desarrollo al ofrecer un tiempo de compilación instantáneo, un servidor de desarrollo veloz y una carga bajo demanda de módulos. Es especialmente popular en combinación con frameworks como Vue.js, React y Preact.

Hablemos de las carpetas de nuestro proyecto

La estructura de carpetas de un proyecto de React utilizando Vite sigue una convención comúnmente utilizada en el ecosistema de React. A continuación, se describe cada una de las carpetas principales y para qué se utilizan:





1. **node_modules**: Esta carpeta se crea automáticamente al instalar las dependencias del proyecto utilizando herramientas como npm (Node Package Manager) o Yarn. Aquí se almacenan todas las bibliotecas y paquetes de terceros que son necesarios para el funcionamiento del proyecto.

2. **public**: En esta carpeta se almacenan los archivos estáticos que se servirán directamente al navegador, como imágenes, archivos CSS, fuentes, etc. Estos archivos se copian directamente en la raíz del directorio de salida al construir el proyecto.

3. **src**: Esta carpeta es donde se ubica el código fuente de la aplicación. A continuación, se describen las subcarpetas y archivos más comunes dentro de la carpeta **src**:

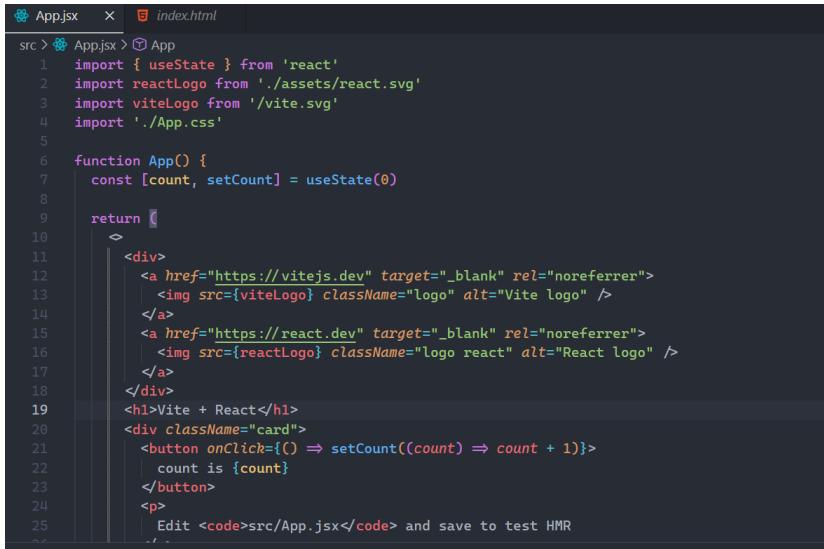
- **index.html**: Es el archivo HTML principal de la aplicación. Aquí se incluirán los puntos de montaje de la aplicación React y se vincularán los archivos CSS o scripts necesarios.
 - **main.js o index.js**: Es el punto de entrada principal de la aplicación. Aquí se configura y se inicia la aplicación React.
 - **components**: Esta carpeta se utiliza para almacenar los componentes de React reutilizables. Cada componente generalmente se coloca en su propio archivo.
 - **pages**: Aquí se pueden almacenar los componentes específicos de cada página de la aplicación. Por lo general, cada archivo en esta carpeta representa una página o una ruta de la aplicación.
 - **assets**: En esta carpeta se pueden almacenar archivos estáticos adicionales, como imágenes, iconos o archivos de datos.
 - **styles**: Aquí se pueden colocar los archivos CSS o preprocesados (como Sass o Less) utilizados en la aplicación.
 - **utils**: En esta carpeta se pueden almacenar funciones de utilidad, helpers o módulos que se utilizan en varias partes del proyecto.
 - **services**: Esta carpeta se utiliza para almacenar módulos de servicios o API que interactúan con servicios externos, como solicitudes HTTP o almacenamiento local.
4. **dist**: Esta carpeta se crea cuando se realiza una compilación o construcción del proyecto. Contiene los archivos estáticos generados y optimizados listos para ser desplegados en producción.
5. **public/index.html**: Este archivo es una plantilla HTML que se utiliza durante el desarrollo y se utiliza para servir la aplicación React. Durante la construcción, este archivo se reemplaza por el archivo **index.html** ubicado en la carpeta **public**.



@hdtoledo

Estas son las carpetas y archivos principales que se encuentran comúnmente en un proyecto de React utilizando Vite. Sin embargo, ten en cuenta que la estructura puede variar dependiendo de las preferencias y necesidades del proyecto, así como de las convenciones establecidas por el equipo de desarrollo.

Conozcamos nuestro código, abriremos App.jsx y lucirá así :



```
App.jsx  X  index.html
src > App.jsx > App
1 import { useState } from 'react'
2 import reactLogo from './assets/react.svg'
3 import viteLogo from '/vite.svg'
4 import './App.css'
5
6 function App() {
7   const [count, setCount] = useState(0)
8
9   return (
10    >
11    <div>
12      <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
13        <img src={viteLogo} className="logo" alt="Vite logo" />
14      </a>
15      <a href="https://react.dev" target="_blank" rel="noreferrer">
16        <img src={reactLogo} className="logo react" alt="React logo" />
17      </a>
18    </div>
19    <h1>Vite + React</h1>
20    <div className="card">
21      <button onClick={() => setCount((count) => count + 1)}>
22        count is {count}
23      </button>
24    </div>
25  > Edit <code>src/App.jsx</code> and save to test HMR
...`
```

La primera parte de nuestro código son las importaciones:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'
```

Aquí, se importa la función **useState** de React, que es un hook para agregar el estado a componentes funcionales. Luego, se importan dos imágenes (**reactLogo** y **viteLogo**) y el archivo **App.css**, que contiene estilos para la aplicación.

Luego tendremos nuestro componente App:



```

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank" rel="noreferrer">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App

```

- Cuando se carga la aplicación, se muestra una página con los logotipos de Vite y React como enlaces. Al hacer clic en estos logotipos, los usuarios serán dirigidos a sus respectivos sitios web (Vite y React) en una nueva pestaña del navegador.
- Debajo de los logotipos, hay un título que dice "Vite + React".
- Luego, hay una tarjeta que contiene un botón y un párrafo. Al hacer clic en el botón, se incrementa el valor del contador **count**.
- Además, hay un párrafo que indica al usuario que edite el archivo **src/App.jsx** y lo guarde para probar la recarga en caliente (HMR - Hot Module Replacement).
- Por último, hay otro párrafo que sugiere a los usuarios hacer clic en los logotipos de Vite y React para obtener más información.

Hagamos una prueba, modificando el H1 y colando otro texto:

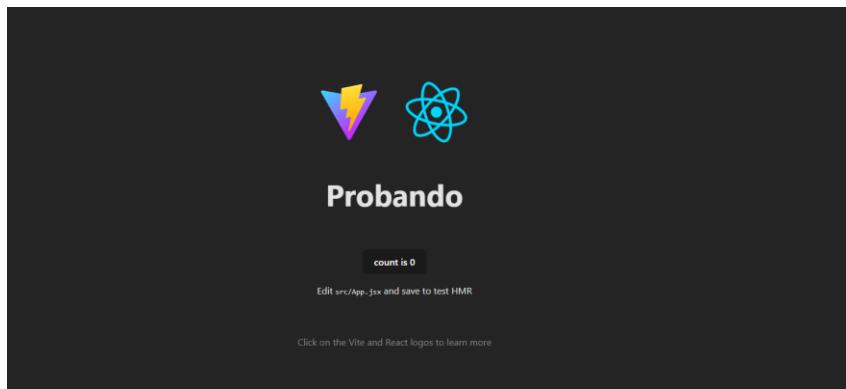
```

15   <a href="https://react.dev" target="_blank" rel="noreferrer">
16     <img src={reactLogo} className="logo react" alt="React logo" />
17   </a>
18 </div>
19 <h1>Probando</h1>
20 <div className="card">
21   <button onClick={() => setCount((count) => count + 1)}>
22     count is {count}
23   </button>

```

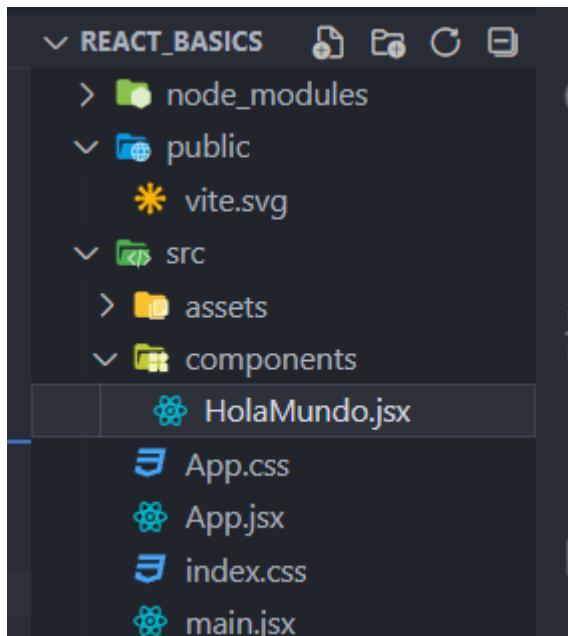


@hdtoledo



De esta manera estamos verificando que podemos cambiar todo lo que encontramos acá y que tenemos prácticamente una plantilla lista para empezar con nuestro proyecto.

Todo componente debe ir en la carpeta SRC, para ello vamos a crear una carpeta que se llame Componentes, allí crearemos nuestro componente HolaMundo.jsx, recordemos que todo componente inicia con mayúsculas por regla general.



Ahora para iniciar nuestro componente, lo primero es realizar la importación de react

```
src > components > HolaMundo.jsx > default
1 import React from "react";
2
```

Luego creamos nuestra función, y acá muy importante los nombres de las funciones deben iniciar con mayúsculas de igual manera que como lo hacemos con los componentes, dentro de nuestra función siempre debe ir un return que es el que nos devolverá lo que le indiquemos dentro.

```

2
3 | function HolaMundo() {
4 |   return(
5 |     <div>
6 |       <h1>Hola a Todos !</h1>
7 |       <h2>Bienvenidos a REACT Basics</h2>
8 |     </div>
9 |
10 }
11

```

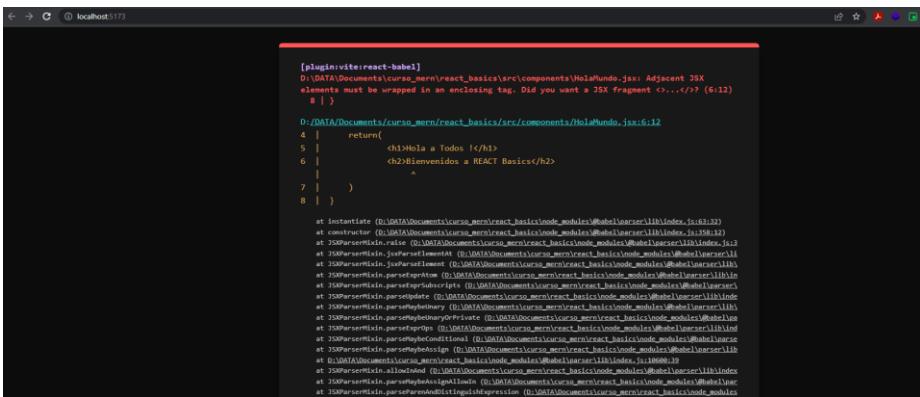
Este return solo podrá devolver un solo objeto a la vez, es decir si colocamos por fuera el h1 y h2 sucedera lo siguiente:

```

src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo() {
4   return[
5     <h1>Hola a Todos !</h1>    Las expresiones JSX deben tener un elemento primario.
6     <h2>Bienvenidos a REACT Basics</h2>
7   ]
8 }

```

Y en nuestro navegador tendremos lo siguiente:



Si no queremos tener un div también podemos utilizar un fragment que simplemente tendrá la siguiente sintaxis `<></>`

```

return(
  <>
    <h1>Hola a Todos !</h1>
    <h2>Bienvenidos a REACT Basics</h2>
  </>
)

```

Y por último siempre debemos exportar nuestro componente para ello hay dos maneras, una al finalizar colocamos `export default NombreComponente`



@hdtoledo

```
11
12  export default HolaMundo|
```

Y la otra forma es simplemente en la parte de arriba donde mencionamos la función le colocamos el export default.

```
❖ HolaMundo.jsx ×
src > components > ❖ HolaMundo.jsx > ...
1  import React from "react";
2
3  export default function HolaMundo() {
4      return(
5          <div>
6              <h1>Hola a Todos !</h1>
7              <h2>Bienvenidos a REACT Basics</h2>
8          </div>
9      )
10 }
```

Siempre debe haber un export default único, no puede haber dos defaults en el mismo componente.

Ahora vamos a nuestro App.jsx y hacemos la importación del componente

```
❖ HolaMundo.jsx    ❖ App.jsx ×
src > ❖ App.jsx > ...
1  import { useState } from 'react'
2  import reactLogo from './assets/react.svg'
3  import viteLogo from '/vite.svg'
4  import './App.css'
5  import HolaMundo from './components/HolaMundo'
6
7  function App() {
```

Lo siguiente es añadir nuestro componente en nuestro código, como si fuera un HTML

```
19      </div>
20      < HolaMundo/>|
21      </>
22
```

De esta manera quedaría nuestro componente



```

❶ HolaMundo.jsx ❷ App.jsx ❸
src > App.jsx > ⚡ App
1 import { useState } from 'react'
2 import reactLogo from './assets/react.svg'
3 import viteLogo from '/vite.svg'
4 import './App.css'
5 import HolaMundo from './components/HolaMundo'
6
7 function App() {
8   const [count, setCount] = useState(0)
9
10  return (
11    <>
12      <div>
13        <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
14          <img src={viteLogo} className="logo" alt="Vite logo" />
15        </a>
16        <a href="https://react.dev" target="_blank" rel="noreferrer">
17          <img src={reactLogo} className="logo react" alt="React logo" />
18        </a>
19      </div>
20      <HolaMundo/>
21    </>
22  )
23}
24
25 export default App
26

```

Y así se vería en nuestra vista:



Modifiquemos un poco nuestro componente HolaMundo, agregamos uno que se llame AdiosMundo y hacemos la exportación sin default:

```

❶ HolaMundo.jsx ❷ App.jsx
src > components > HolaMundo.jsx > ⚡ AdiosMundo
1 import React from "react";
2
3 export default function HolaMundo() {
4   return(
5     <div>
6       <h1>Hola a Todos !</h1>
7       <h2>Bienvenidos a REACT Basics</h2>
8     </div>
9   )
10 }
11
12 export function AdiosMundo(){
13   return(
14     <div>
15       <h3>Adios a Todos!</h3>
16     </div>
17   )
18 }

```

Para hacer la importación de este componente debemos agregarlo no en una línea aparte sino en la misma importación que tenemos de HolaMundo, agregando una coma y utilizando las llaves con el nombre del componente dentro:

```
4  import './App.css'
5  import HolaMundo, { AdiosMundo } from './components/HolaMundo'
6
```

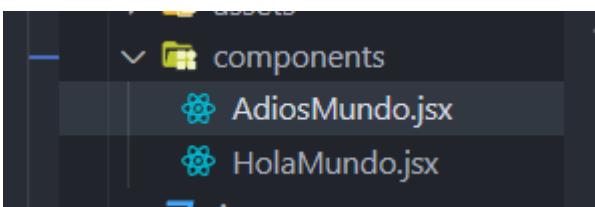
Y de esta manera procedemos a añadir nuestro nuevo componente:

```
18    </a>
19    </div>
20    < HolaMundo/>
21    < AdiosMundo/>
22  </>
23 }
24 }
```

Y nos quedaría de la siguiente manera:



Lo ideal es que generemos componentes separados para no causar equivocaciones, ahora crearemos un componente aparte que se llame AdiosMundo:



Y dentro de nuestro `.jsx` colocaremos de una manera mas breve el código, utilizando `rfce` (que es una abreviación de react functional component export) quedando así:



```
src > components > AdiosMundo.jsx
```

1 rfce	
└─ rfce	reactFunctionalExportComponent
└─ rfce	reactFunctionalExportComponent
└─ rfcredux	reactFunctionalComponentRedux

Para que nos salga debemos tener la extensión reactjs code snippets.

```
src > components > AdiosMundo.jsx > AdiosMundo
```

```
1 import React from 'react'
2
3 function AdiosMundo() {
4   return (
5     <div>AdiosMundo</div>
6   )
7 }
8
9 export default AdiosMundo
```

De esta manera simplificaremos el código y nos creara nuestro componente con su export, hacemos la importación en App.jsx y procedemos a revisarlo en nuestro navegador

```
src > App.jsx > ...
1 import { useState } from 'react'
2 import reactLogo from './assets/react.svg'
3 import viteLogo from '/vite.svg'
4 import './App.css'
5 import HolaMundo from './components/HolaMundo'
6 import AdiosMundo from './components/AdiosMundo'
7
8
9 function App() {
10   const [count, setCount] = useState(0)
11
12   return (
13     <>
14       <div>
15         <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
16           <img src={viteLogo} className="logo" alt="Vite logo" />
17         </a>
18         <a href="https://react.dev" target="_blank" rel="noreferrer">
19           <img src={reactLogo} className="logo react" alt="React logo" />
20         </a>
21       </div>
22       < HolaMundo/>
23       < AdiosMundo/>
24     </>
25   )
26
27
28 export default App
29
```



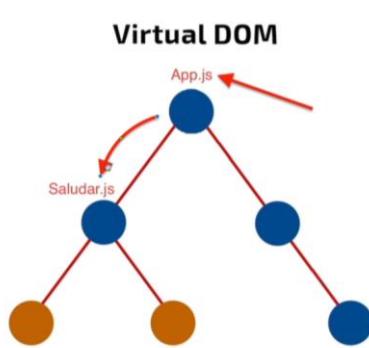
@hdtoledo



No olvidemos eliminar el **AdiosMundo** anterior.

¿Qué son los props en react?

Los Props en react es una zona en donde pasamos información de un componente a otro:

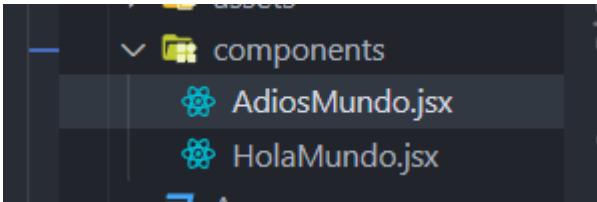


Supongamos que nuestra **app** empieza por **app.js** y es el componente padre, este tiene componentes hijos como lo es **saludar.js**, en el ejercicio imaginamos que en **app.js** nos muestra un saludo: Hola, (**nombreUsuario**) y esto nos mostraría un saludo con el nombre del usuario, este mismo se pasara a **saludar.js** y tiene acceso a la información del usuario que estamos pasando por allí, es decir que la información de este componente padre no termina en **saludar.js** sino que continua vigente y se puede utilizar en los demás componentes.

Los **props** permiten pasar información entre componentes y esta información puede venir de diferentes componentes.

Pasando Props básicos entre componentes

En nuestro proyecto de ejemplo actual tenemos dos componentes:



Que se están visualizando en app.jsx

```
src > App.jsx > App
8
9  function App() {
10    const [count, setCount] = useState(0)
11
12    return (
13      <>
14        <div>
15          <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
16            <img src={viteLogo} className="logo" alt="Vite logo" />
17          </a>
18          <a href="https://react.dev" target="_blank" rel="noreferrer">
19            <img src={reactLogo} className="logo react" alt="React logo" />
20          </a>
21        </div>
22        <HolaMundo/>
23        <AdiosMundo/>
24      </>
25    )
26  }
27
28 export default App
```

Si revisamos estos componentes son estáticos y no traen **props** lo mismo sucede con que no son reutilizables ya que si duplicamos uno de ellos nos repetirá el contenido



Para el ejercicio vamos a pasar una propiedad tipo texto a nuestro componente HolaMundo, y de esta manera podremos hacer que el componente sea reutilizable.



Vamos a modificar nuestro **HolaMundo** que actualmente se encuentra asi:

```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo() {
4
5   return(
6     <div>
7       <h1>Hola a Todos !</h1>
8       <h2>Bienvenidos a REACT Basics</h2>
9     </div>
10  );
11}
```

Y lo vamos a dejar de la siguiente manera:

```
src > components > HolaMundo.jsx > ...
1 import React from "react";
2
3 export default function HolaMundo() {
4   return (
5     <div>
6       <h1>Bienvenido Usuario</h1>
7     </div>
8   );
9 }
10
```



Ahora para entender cómo vamos a pasar las **props** en nuestra función vamos a agregar la palabra **props** dentro de los parámetros de **HolaMundo**



```

src > components > HolaMundo.jsx > ...
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props)
5   return (
6     <div>
7       <h1>Bienvenido Usuario</h1>
8     </div>
9   );
10 }
11

```

Vamos a revisar en nuestro navegador como está pasando los **props** a través de la consola

The screenshot shows a browser window with developer tools open. The page content includes two logos (Vite and React) and the text "Bienvenido Usuario". Below the page content, the developer tools' "Console" tab is active, showing the output of the console.log(props) statement. The output is an empty object {}, indicating that the props object is currently empty.

Acá podemos observar que nos llega un objeto completamente vacío, lo que vamos a hacer es pasarle una propiedad a través de nuestro componente, es decir vamos a editar nuestro **App.jsx** y lo dejamos de la siguiente manera:

```

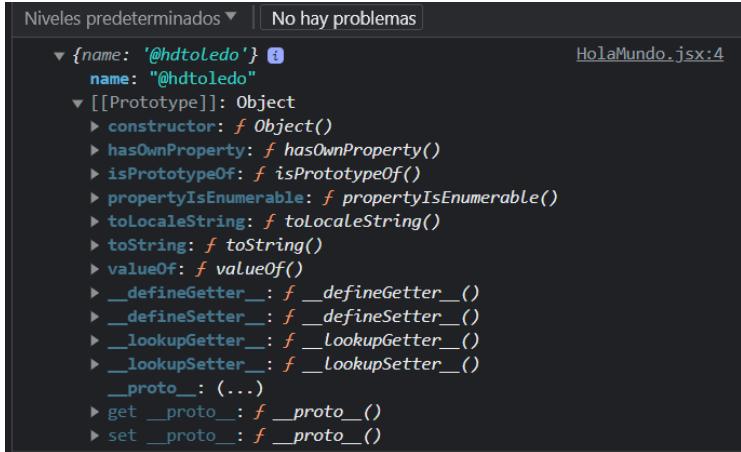
src > App.jsx > ...
8
9 function App() {
10   const [count, setCount] = useState(0)
11
12   return (
13     <>
14       <div>
15         <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
16           <img src={viteLogo} className="logo" alt="Vite logo" />
17         </a>
18         <a href="https://react.dev" target="_blank" rel="noreferrer">
19           <img src={reactLogo} className="logo react" alt="React logo" />
20         </a>
21       </div>
22       < HolaMundo name="@hdtoledo" />
23       < AdiosMundo />
24     </>
25   )
26 }
27
28 export default App
29

```



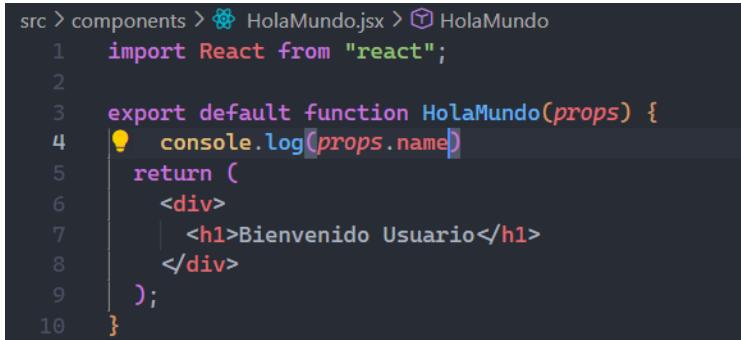
@hdtoledo

Pasamos la propiedad **name** con un **string** y al verlo reflejado en el navegador observaremos lo siguiente:



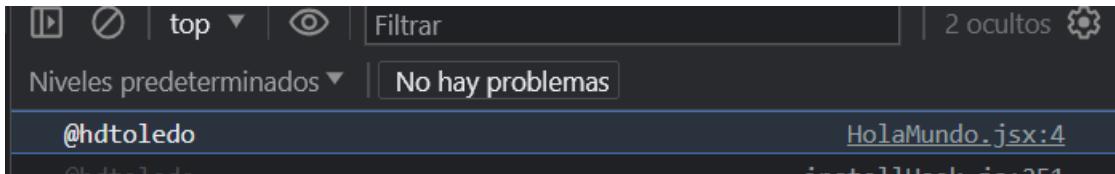
The screenshot shows the Chrome DevTools Elements tab with the title "Niveles predeterminados" and "No hay problemas". A dropdown menu is open, showing the properties of the "name" prop. The properties listed are: name: '@hdtoledo', name: '@hdtoledo', [[Prototype]], constructor, hasOwnProperty, isPrototypeOf, propertyIsEnumerable, toLocaleString, toString, valueOf, __defineGetter__, __defineSetter__, __lookupGetter__, __lookupSetter__, __proto__, get __proto__, and set __proto__. The file "HolaMundo.jsx:4" is indicated next to the first two "name" properties.

Ahora para acceder solamente a la propiedad del objeto hacemos lo siguiente:



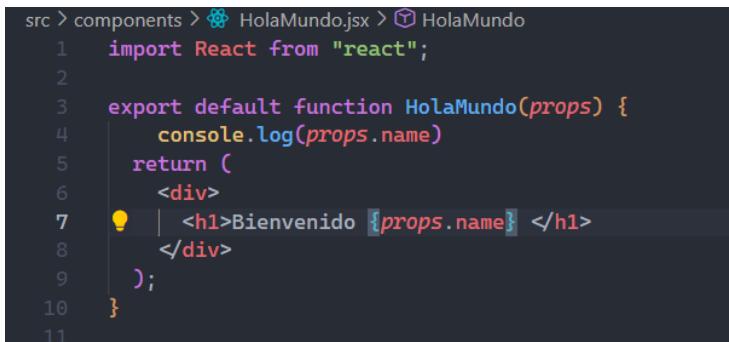
```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props.name)
5   return (
6     <div>
7       <h1>Bienvenido Usuario</h1>
8     </div>
9   );
10 }
```

Simplemente añadimos `props.name` para poder acceder y lo que vemos reflejado en la consola



The screenshot shows the Chrome DevTools Console tab with the title "Niveles predeterminados" and "No hay problemas". The output of the console.log statement is shown as "@hdtoledo". The file "HolaMundo.jsx:4" is indicated next to the output. The browser address bar shows "localhost:3001/HolaMundo" and the port "3051".

Nuestra propiedad **name** dentro de **props**, ahora simplemente utilizamos las llaves para poder llamar la propiedad `name` dentro y nos quedaría así:



```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props.name)
5   return (
6     <div>
7       <h1>Bienvenido {props.name} </h1>
8     </div>
9   );
10 }
```



Y en nuestro navegador:



De esta manera ya estamos utilizando correctamente nuestro componente para pasarle props y tambien podemos duplicarlo y reutilizarlo, haciendo lo siguiente:

```
    </div>
    < HolaMundo name="@hdtoledo" />
    < HolaMundo />

    < AdiosMundo />
</>
)
```

Si lo dejamos solo, nuestro componente nos mostrara lo siguiente



Quiere decir que podemos pasar otra propiedad de name:



```
21      |     </div>
22      |     < HolaMundo name="@hdtoledo" />
23      |     < HolaMundo name="Diana" />
24
25      |     < AdiosMundo />
26  
```

Ahora en nuestro navegador observamos



También podemos pasar más propiedades dentro, vamos a modificar un poco nuestro HolaMundo:

```
src > components > HolaMundo.jsx > ...
1  import React from "react";
2
3  export default function HolaMundo(props) {
4    console.log(props.name)
5    return (
6      <div>
7        |   <h1>Bienvenido, {props.name} tienes {props.edad} años de edad.</h1>
8        </div>
9    );
10  }
11  
```

Acá podemos observar que vamos a pasar otra propiedad que se llamará edad y la vamos a colocar dentro de nuestro mensaje.

En app.jsx quedaría así:

```
21      |     </div>
22      |     < HolaMundo name="@hdtoledo" edad="37" />
23      |     < AdiosMundo />
24  
```



Y en nuestro navegador quedaría así:



Así de esta manera comprendemos que en un componente podemos tener diferentes props que podemos pasarle y que podemos reutilizar.

Pasando Variables Objetos entre componentes por los Props

Imaginemos que tenemos 50 datos de personas para pasar a través de nuestro componente, entonces tendríamos que llenar 50 veces nuestro componente con esos datos, lo cual sería una labor bastante tediosa, para ello lo que podemos realizar es pasar variables y objetos dentro de nuestro componente para implementar una solución.

Veamos como pasamos a través de variables el nombre y la edad, modificando de la siguiente manera nuestro `app.jsx`:

```
src > App.jsx > App
  8
  9  function App() {
10
11    const userName = "HD"
12    const edad = 37
13
14    return [
15      <>
16        <div>
17          <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
18            <img src={viteLogo} className="logo" alt="Vite logo" />
19          </a>
20          <a href="https://react.dev" target="_blank" rel="noreferrer">
21            <img src={reactLogo} className="logo react" alt="React logo" />
22          </a>
23        </div>
24        < HolaMundo name={userName} edad={edad} />
25        < AdiosMundo />
26      </>
27    ]
28  }
29
30  export default App
31
```



Cargamos en una **const** las variables **userName** y **edad**, asignamos los valores y por último modificamos nuestras propiedades con las variables utilizando las llaves para poder traer la información de esta manera nos saldría en nuestro navegador:



Para hacer unos pequeños ajustes, vamos a modificar nuestro `console.log` de `HolaMundo` para que se quede solo con los props:

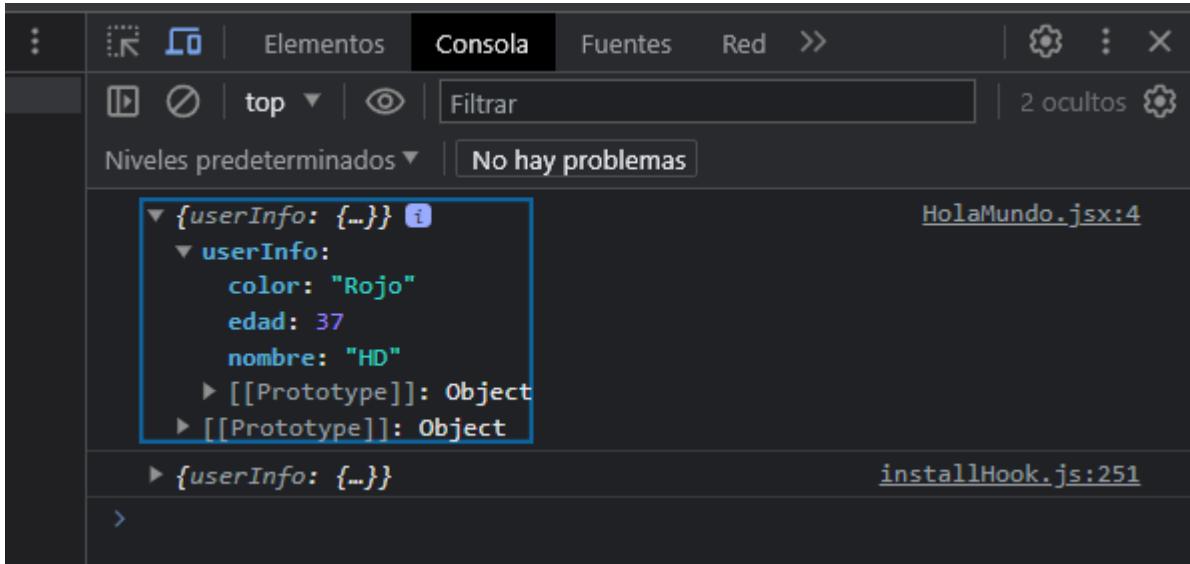
```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props)
5   return (
6     <div>
7       <h1>Bienvenido, {props.name} tienes {props.edad} años de edad.</h1>
8     </div>
9   );
10 }
11
```

Ahora vamos a crear un objeto con unas propiedades del usuario para empezar a utilizarlas mejor dentro de nuestro `app.jsx`:

```
src > App.jsx > ...
8
9 function App() {
10
11   const user = {
12     nombre: "HD",
13     edad: 37,
14     color: "Rojo"
15   }
16
17   return (
18     <>
19       <div>
20         <a href="https://vitejs.dev" target="_blank" rel="noopener">
21           <img src={viteLogo} className="logo" alt="Vite logo" />
22         </a>
23         <a href="https://react.dev" target="_blank" rel="noopener">
24           <img src={reactLogo} className="logo react" alt="React logo" />
25         </a>
26       </div>
27       < HolaMundo userInfo={user} />
28       < AdiosMundo />
29     </>
30   )
31 }
32
33 export default App
```



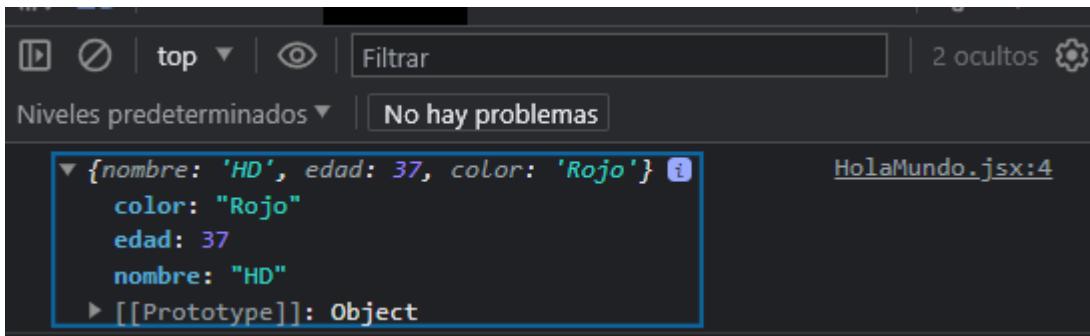
Observemos que hemos creado un **objeto** con las propiedades de **nombre**, **edad**, **color** y que además de ello dentro de nuestro componente **HolaMundo** estamos pasando la información de nuestro usuario, para verificarlo lo podemos ver en nuestro navegador en la consola:



Ahora modificamos un poco el `console.log` de nuevo para pasar `userInfo` allí:

```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props.userInfo)
5   return (
6     <div>
7       <h1>Bienvenido, {props.name} tienes {props.edad} años de edad.</h1>
8     </div>
9   );
10 }
11
```

De esta manera accedemos a la información también pero ya directamente al objeto



Si queremos acceder solo al nombre del objeto modificamos de la siguiente manera:



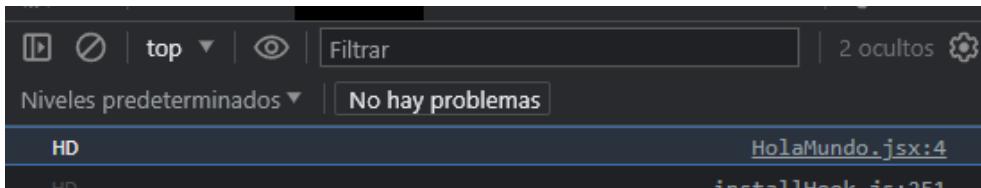
@hdtoledo

```

src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props.userInfo.nombre)
5   return (
6     <div>
7       <h1>Bienvenido, {props.name} tienes {props.edad} años de edad.</h1>
8     </div>
9   );
10 }

```

Ingresamos directamente al nombre del objeto y saldría así en nuestra consola:



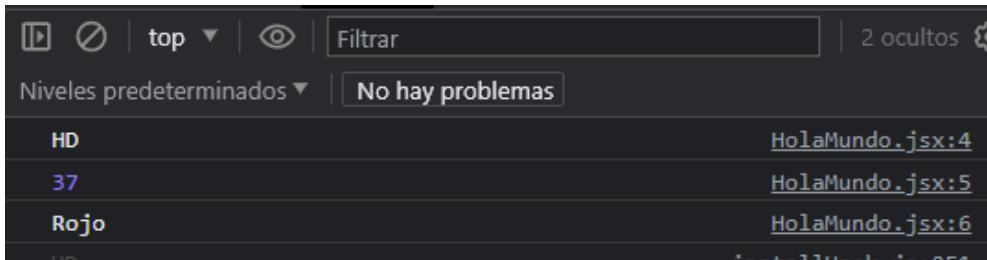
Si colocamos en consola las demás propiedades

```

3   export default function HolaMundo(props) {
4     console.log(props.userInfo.nombre)
5     console.log(props.userInfo.edad)
6     console.log(props.userInfo.color)
7
8     return (
9       <div>

```

Y de esta manera en consola observaremos los datos



Ahora vamos a realizar unos cambios en nuestro **HolaMundo**:

```

src > components > HolaMundo.jsx > ...
1 import React from "react";
2
3 export default function HolaMundo(props) {
4   console.log(props.userInfo.nombre)
5   console.log(props.userInfo.edad)
6   console.log(props.userInfo.color)
7
8   return (
9     <div>
10       <p>Bienvenido, <b>{props.userInfo.nombre}</b> tienes <b>{props.userInfo.edad}</b> años de edad y tu
11       color favorito es <b>{props.userInfo.color}</b>.</p>
12     </div>
13   );
14 }

```

Cambiamos por un párrafo y ajustamos el mensaje que estamos mostrando mediante las propiedades del objeto **userInfo**, quedándonos de la siguiente manera:





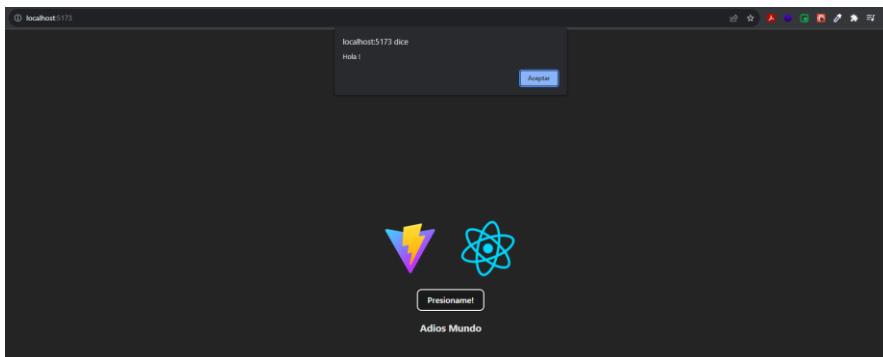
Y de esta manera estamos pasando variables y objetos entre los componentes.

Pasando Funciones entre componentes por los Props

Para pasar las funciones dentro de los componentes e incluir los props vamos a hacer un ejercicio sencillo, realizamos una función tipo flecha para que un botón que simplemente al hacer clic se nos muestre un alert:

```
src > components > HolaMundo.jsx > HolaMundo
1  import React from "react";
2
3  export default function HolaMundo(props) {
4
5    const mensaje = () => {
6      alert('Hola !')
7    }
8
9    return (
10      <div>
11        <button onClick={mensaje}>Presioname!</button>
12      </div>
13    );
14
15 }
```

De esta manera lo visualizamos



Ahora vamos a implementar que al presionar nuestro botón se muestren las props que vamos a enviarle, para ello modificamos nuestro HolaMundo:



```

src > components > HolaMundo.jsx > ...
1  import React from "react";
2
3  export default function HolaMundo(props) {
4    return (
5      <div>
6        <button onClick={() => props.saludarFn(props.userInfo.nombre)}>Presioname!</button>
7      </div>
8    );
9  }
10

```

Vamos a colocar que al hacer clic en nuestro botón se ejecute la función **saludarFn** con el nombre del objeto **userInfo** que tenemos en nuestro **app.jsx**, y dentro de **app.jsx** vamos a crear la función, notemos que para poder ejecutarla dentro de **onClick** realizamos una función tipo flecha para que no se ejecute automáticamente, sino que espere a que demos clic sobre el botón. Y modificamos nuestro **app.jsx** de la siguiente manera:

```

src > App.jsx > App
9  function App() {
10
11    const user = {
12      nombre: "HD",
13      edad: 37,
14      color: "Rojo"
15    }
16
17    const saludarFn = name => {
18      alert("Hola, " + name)
19    }
20
21    return (
22      <>
23        <div>
24          <a href="https://vitejs.dev" target="_blank" rel="noreferrer">
25            <img src={viteLogo} className="logo" alt="Vite logo" />
26          </a>
27          <a href="https://react.dev" target="_blank" rel="noreferrer">
28            <img src={reactLogo} className="logo react" alt="React logo" />
29          </a>
30        </div>
31        <HolaMundo userInfo={user} saludarFn={saludarFn} />
32        <AdiosMundo/>
33      </>
34    )
35  }
36
37  export default App
38

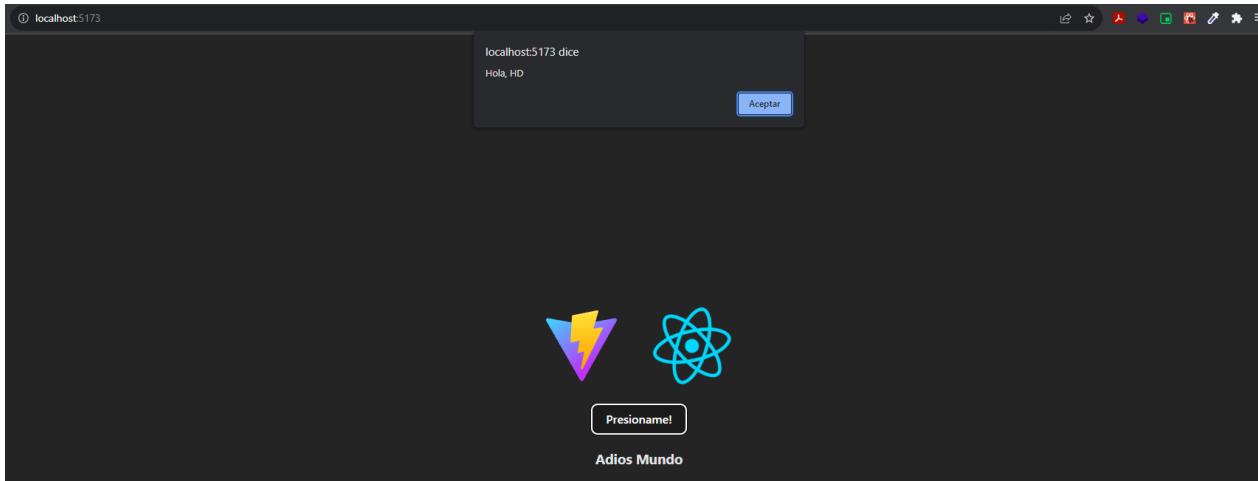
```

Cargamos en una función tipo flecha que se llamará **saludarFn** que va a traer el **name** y lo va a mostrar dentro de un **alert**, además de que pasamos **saludarFn** a través del componente para que se ejecute con los **props** que ya le hemos definido, de esta manera mandamos la función al componente hijo, la



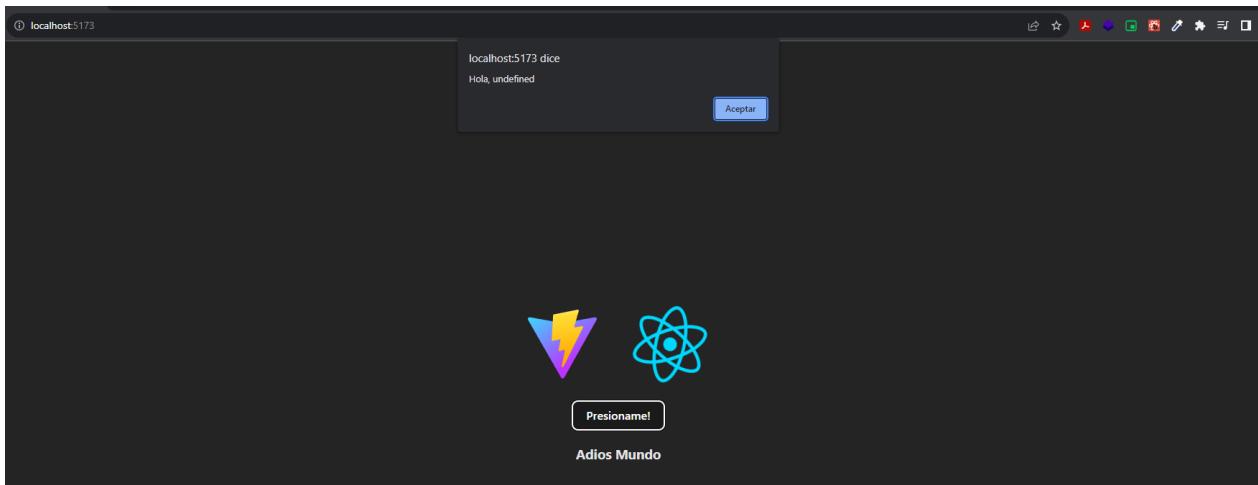
@hdtoledo

ejecutamos y le estamos enviando al componente padre el usuario, de esta forma enviamos props entre ambos tanto al componente padre como al hijo y viceversa.



Ahora que sucede si nuestro objeto no tiene un nombre definido ¿? Pues simplemente nos mostrara undefined:

```
src > ⚛ App.jsx > ↗ App > 📂 user
  9   function App() {
10
11     const user = {
12       |
13       edad: 37,
14       color: "Rojo"
15     }
16
17     const saludarEn = name => {
```



@hdtoledo

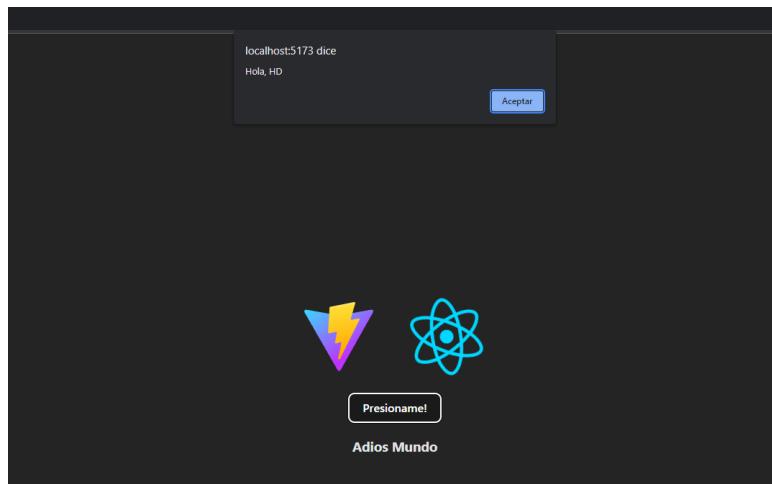
El uso de la asignación por destructuring

Vamos a realizar una **const** dentro de **HolaMundo** con los datos de nuestro **props**, en este vamos a pasarle **userInfo** y **saludarFn**, para ello mostramos a través de consola lo que estamos trayendo:

```
src > components > HolaMundo.jsx > ...
1 import React from "react";
2
3 export default function HolaMundo(props) {
4
5   const {userInfo, saludarFn} = props
6
7   return (
8     <div>
9       <button onClick={() => props.saludarFn(userInfo.nombre)}>Presioname!</button>
10      </div>
11    );
12 }
```

Y acá podemos verificar que dentro de nuestro botón no es necesario acceder a **props**, sino que vamos a ingresar a **userInfo** directamente.

Y en nuestro navegador podemos verificar que la información se nos muestra:



También podemos comprobar los datos a través de un `console.log` para que nos los muestre:

```
4
5   const {userInfo, saludarFn} = props
6   console.log(userInfo)
7
8   return (
```

localhost:5173 - HolaMundo.js

Elementos Consola Fuentes Red Rendimiento Memoria >

top ▾ Filtrar Niveles predeterminados ▾ 2 ocultos

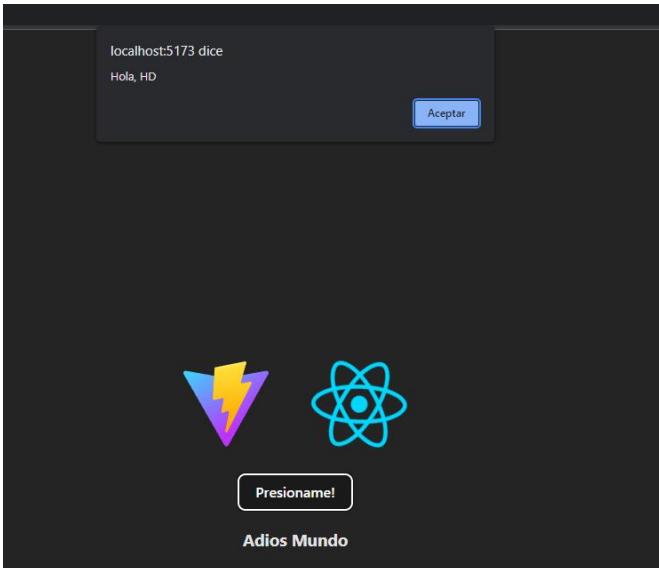
No hay problemas

▶ {nombre: 'HD', edad: 37, color: 'Rojo'} HolaMundo.jsx:6



De la misma manera que estamos accediendo directamente a la información, lo podemos hacer con **saludarFn** sin necesidad de acceder a **props**.

```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4
5   const {userInfo, saludarFn} = props
6   console.log(userInfo)
7
8   return (
9     <div>
10    |   <button onClick={() => saludarFn(userInfo.nombre)}>Presioname!</button>
11   </div>
12 );
13 }
```



El destructuring lo podemos aplicar directamente sobre nuestros datos asignando directamente sobre una **const** los datos de **userInfo**

```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4
5   const {userInfo, saludarFn} = props
6   const { nombre } = userInfo
7
8   return (
9     <div>
10    |   <button onClick={() => saludarFn(nombre)}>Presioname!</button>
11   </div>
12 );
13 }
```

Sacamos la información de una variable a otra **const** y así realizamos el destructuring para acceder de una manera mas eficiente y nos ayuda a implementar unas buenas practicas.



Props por default

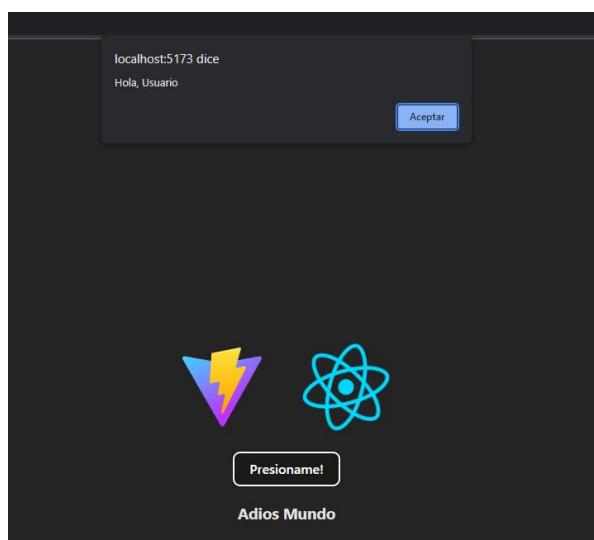
También podemos asignar **props** por default directamente, si el usuario no trae un nombre debemos asignarle un valor default, para ello vamos a colocarle ese valor de la siguiente manera a **nombre**:

```
src > components > HolaMundo.jsx > HolaMundo > nombre
  1 import React from "react";
  2
  3 export default function HolaMundo(props) {
  4
  5   const {userInfo, saludarFn} = props
  6   const [ nombre = "Usuario" ] = userInfo
  7
  8   return (
  9     <div>
 10       <button onClick={() => saludarFn(nombre)}>Presioname!</button>
 11     </div>
 12   );
 13 }
```

Para comprobarlo simplemente eliminamos la propiedad de **nombre** en nuestro **app.jsx**

```
src > ⚙ App.jsx > 📂 App > 📁 user          src > ⚙ App.jsx > 📂 App > 📁 user
  9  function App() {                      9  function App() {
 10    |   const user = {                  10
 11    |     nombre: "HD",              11
 12    |     edad: 37,                 12
 13    |     color: "Rojo"            13
 14    |   }                          14
 15  }                                15
 16
```

Y de esta manera al presionar el botón observamos que ya viene el valor por default:



Template Strings o Template Literals

Los "template literals" o "template strings" son una característica introducida en ECMAScript 6 (también conocido como ES6) que permite crear cadenas de texto de una manera más flexible y legible en JavaScript. Antes de la introducción de los template literals, la concatenación de cadenas en JavaScript solía hacerse utilizando el operador `+` o mediante la función `concat()`, lo que a menudo resultaba en código menos legible y más propenso a errores.

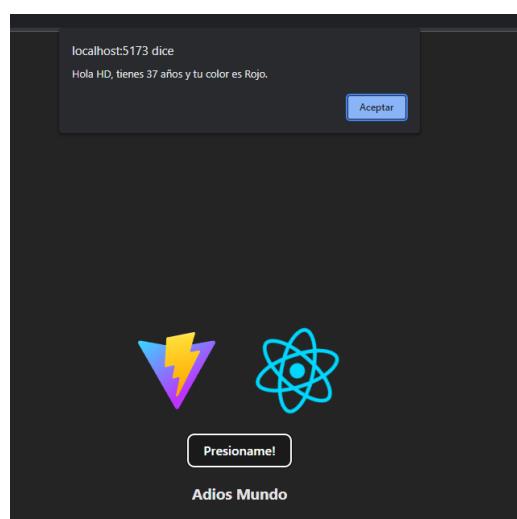
Sintaxis: Los template literals se definen utilizando comillas invertidas (backticks) ``...``, en lugar de comillas simples o dobles, y allí para utilizar las variables abrimos a través del signo de **pesos \$** y **llaves {}** y dentro colocamos la variable, vamos a realizar un pequeño cambio en nuestro código, pasamos primero unas props extras en `app.jsx` agregamos `edad` y `color` a nuestra `const`.

```
src > App.jsx > App
14   color: "Rojo"
15 }
16
17 const saludarFn = (nombre, edad, color) => {
18   alert(`Hola ${nombre}, tienes ${edad} años y tu color es ${color}.`)
19 }
20
21 return (
22   <>
```

Y en nuestro `alert` modificamos con los **backticks** el mensaje e implementamos `${variable}`, de esta manera pasamos a modificar ahora **HolaMundo**

```
src > components > HolaMundo.jsx > HolaMundo
1 import React from "react";
2
3 export default function HolaMundo(props) {
4
5   const {userInfo, saludarFn} = props
6   const { nombre = "Usuario", edad = 0, color = "Ninguno" } = userInfo
7
8   return (
9     <>
```

Ahora pasamos directamente con el destructuring nuestras variables y les asignamos un valor por default, de esta manera al ir al navegador observaremos como pasamos todos los datos en nuestro `alert`.



Hook de estado – useState

El hook useState es utilizado para crear variables de estado, quiere decir que su valor es dinámico, que este puede cambiar en el tiempo y eso requiere una re-renderización del componente donde se utiliza

Recibe un parámetro:

- El valor inicial de nuestra variable de estado.

Devuelve un array con dos variables:

- En primer lugar, tenemos la variable que contiene el valor
- La siguiente variable es una función set, requiere el nuevo valor del estado, y este modifica el valor de la variable que anteriormente mencionamos
- Cabe destacar que la función proporciona como parámetro el valor actual del propio estado.
Ex: setisOpen(isOpen => !isOpen)

En este ejemplo mostramos como el valor de count se inicializa en 0, y también se renderiza cada vez que el valor es modificado con la función setCount en el evento onClick del button:

```
import { useState } from 'react'

function Counter() {
  const [count, setCount] = useState(0)

  return (
    <>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count => count + 1)}>Aumentar</button>
    </>
  )
}
```

Ahora vamos a colocarlo a prueba, para ello recordemos que tenemos un componente **AdiosMundo**, dentro de este vamos a realizar lo siguiente:

```
src > components > AdiosMundo.jsx > ...
1  import React from 'react'
2
3  function AdiosMundo() {
4
5    const encenderApagar = () => {
6      alert('Encender / Apagar')
7    }
8
9    return (
10      <div>
11        <h3>El carro esta: Encendido</h3>
12        <button onClick={encenderApagar}>Encender / Apagar</button>
13      </div>
14    )
15  }
16
17  export default AdiosMundo
```

Creamos una **const** función tipo flecha que se llamará **encenderApagar** y nos mostrará un **alert**, y vamos a colocar un **h3** con un mensaje y un **botón** que al ser presionado llamará la función tipo flecha, lo que queremos realizar será que al presionar el botón nos cambie el estado de encendido a apagado, para ello vamos a importar **useState** de la siguiente manera:



```
src > components > AdiosMundo.jsx > ...
1 import React, { useState } from 'react'
2 |
3 function AdiosMundo() {
4 }
```

Y ahora almacenamos en una **const** en forma de array los valores que vamos a pasar y estos los utilizamos a través de **useState** estableciendo un valor booleano

```
src > components > AdiosMundo.jsx > AdiosMundo
1 import React, { useState } from 'react'
2
3 function AdiosMundo() {
4
5   const [stateCar, setStateCar] = useState(false)
6
7   const encenderApagar = () => {
8     alert('Encender / Apagar')
9   }
10 }
```

Y vamos ahora a colocar una condición sobre nuestro mensaje que nos muestre el texto encendido o apagado:

```
10
11   return (
12     <div>
13       <h3>El carro esta: { stateCar ? "Encendido" : "Apagado" }</h3>
14       <button onClick={encenderApagar}>Encender / Apagar</button>
15     </div>
16   )
17 }
18 }
```

Al estar en false nuestro mensaje será Apagado, pero si modificamos a true nos aparecerá encendido:



Para cambiar nuestro estado utilizaremos la función **setStateCar** en donde nos permitirá cambiar su valor, para ello modificamos la función **encenderApagar**:

```
6
7   const encenderApagar = () => {
8     setStateCar(!stateCar)
9   }
10 }
```

El valor que nos recibirá será el contrario al que nosotros establecemos, es decir si esta true, pasara un valor false y viceversa, para comprobarlo presionaremos el botón y veremos como se actualiza.



Hook de Estado – useEffect

El hook useEffect se usa para ejecutar código cuando se renderiza el componente o cuando cambian las dependencias del efecto.

Recibe dos parámetros:

- La función que se ejecutará al cambiar las dependencias o al renderizar el componente.
- Un array de dependencias. Si cambia el valor de alguna dependencia, ejecutará la función.

En este ejemplo mostramos un mensaje en consola cuando carga el componente y cada vez que cambia el valor de count:

```
import { useEffect, useState } from 'react'

function Counter() {
  const [count, setCount] = useState(0)

  useEffect(() => {
    console.log('El contador se ha actualizado')
  }, [count])

  return (
    <>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Aumentar</button>
    </>
  )
}
```

Ahora para aplicarlo vamos a crear un contador que lleve la cuenta de cuantos clics hemos dado al botón de encender / apagar, para ello creamos una **const** en donde pasamos **contar** y **setContar**

```
5 |   const [stateCar, setStateCar] = useState(false)
6 |   const [contar, setContar] = useState(0)
7 |
8 |   const encenderApagar = () => {
```

Y ahora lo que vamos a realizar es crear un **useEffect**, esta es su sintaxis:

```
8 |
9 |   useEffect(() => {
10 |
11 |     }, [])
12 |
```

Verificamos que se esté importando:

```
src > components > AdiosMundo.jsx > AdiosMundo
1  import React, { useEffect, useState } from 'react'
2
3  function AdiosMundo()
```

Vamos a agregar antes de usar el **useEffect** un **h4** que nos diga total de **clics** y lo asignamos con **contar**:

```
18 |   <div>
19 |     <h3>El carro esta: { stateCar ? "Encendido" : "Apagado" }</h3>
20 |     <h4>Total de clics: {contar}</h4>
21 |     <button onClick={encenderApagar}>Encender / Apagar</button>
22 |   </div>
```

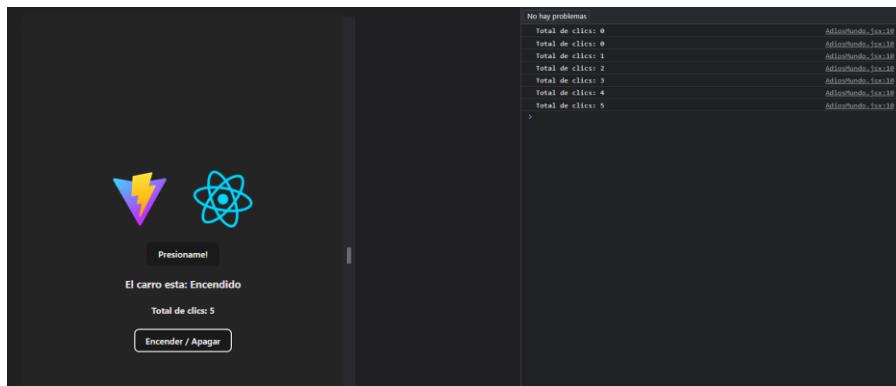
Y ahora vamos a utilizar nuestro **useEffect** de la siguiente manera:

```
8
9  useEffect(() => {
10    console.log(`Total de clics: ${contar}`)
11  }, [contar])
12
```

Mediante un **console.log** llevaremos el conteo también y asignamos contar al **array** para que luego pueda ser utilizado por nuestra función **encenderApagar** que le agregamos la función de **setContar** que sumara la cantidad de clics dados:

```
13 const encenderApagar = () => {
14   setStateCar(!stateCar)
15   setContar(contar + 1)
16 }
17
```

Al revisar en nuestro navegador ya tendremos el conteo tanto en nuestro **h4** como en consola:

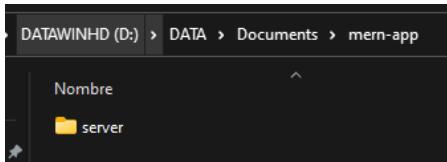


Y cada vez que se actualiza la página el contador vuelve a 0.



Creación del proyecto

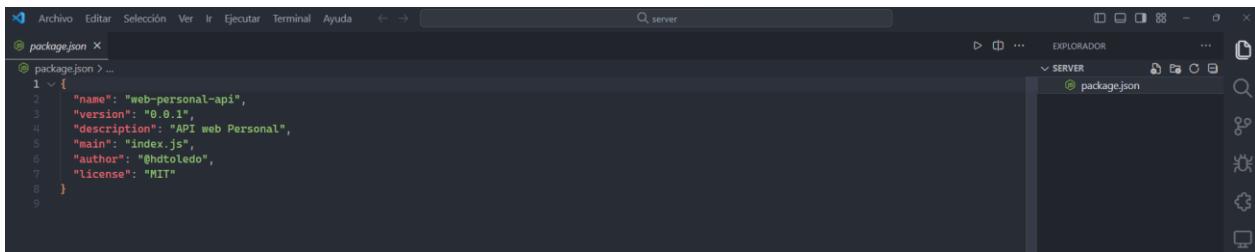
Vamos a iniciar creando nuestro proyecto base, para ello vamos a trabajar en el **frontend** y en el **backend**, también podríamos mencionarlos como el **cliente** y el **server**, lo primero es crear la carpeta principal, en mi caso trabajare en la carpeta **/mern-app** y allí creare una carpeta **server**:



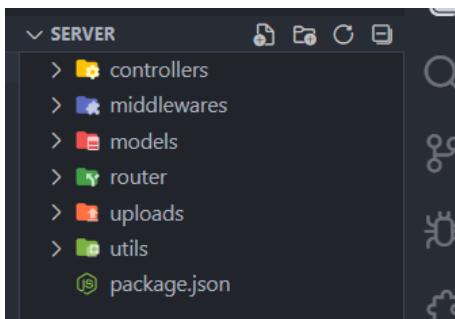
Abrimos nuestra terminal y vamos a entrar a esta carpeta, y colocamos el comando **yarn init**

```
pwsh D:\DATA\Documents\mern-app
> cd .\server\
pwsh D:\DATA\Documents\mern-app\server
> yarn init
yarn init v1.22.19
question name (server): web-personal-api
question version (1.0.0): 0.0.1
question description: API web Personal
question entry point (index.js):
question repository url:
question author: @hdtolledo
question license (MIT):
question private:
success Saved package.json
Done in 58.19s.
```

Aquí nos saldrán una serie de preguntas para iniciar nuestro **package.json**, llenamos cada una de ellas, procedemos a abrirlo en nuestro editor **VS code** y nos mostrara el **package.json**



Ahora procederemos a crear parte de la estructura de nuestro proyecto, es decir creamos las carpetas en la raíz de server:



CREANDO UN CLUSTER MONGO ATLAS

The screenshot shows the MongoDB Atlas landing page. On the left, there's a large text area with the heading "For the next generation of intelligent applications." Below it, a subtext reads: "Build applications on the industry's first developer data platform. From AI-powered and event-driven apps to edge use cases and search, build fast and at the scale users demand." At the bottom, there are two buttons: "Start Free" and "Learn More →". On the right, a modal window is open with the title "Connecting to: mongodb://cluster0.ab123.mongodb.net/myFirstDb" and the message "Using MongoDB: 7.0". At the bottom of the modal, there are "Confirm" and "Cancel" buttons.

Iremos a la página oficial de [mongodb](#) en donde crearemos un **cluster** para nuestra base de datos NOSQL, para ellos nos registramos dentro de mongo y allí veremos lo siguiente:

The screenshot shows the MongoDB Atlas interface under the "Database Deployments" tab. It features a sidebar with sections like "DEPLOYMENT", "SERVICES", and "SECURITY". The main area has a "Create a database" button with the sub-instruction "Choose your cloud provider, region, and specs." Below it, a note says "Once your database is up and running, see regions on existing MongoDB databases into Atlas with our Live Migration service." There's also a "Build a Database" button.

Procedemos a darle en **Build Database**, y allí nos mostrara lo siguiente:

The screenshot shows the "Deploy your database" configuration page. It displays three plan options: "M10 \$0.08/hour", "SERVERLESS \$0.10/1M reads", and "M0 FREE". Each plan includes a brief description and resource details. The "M0" plan is highlighted with a blue border. A "Compare Features" button is located at the bottom.

Plan	Cost	Description
M10	\$0.08/hour	For production applications with sophisticated workload requirements.
SERVERLESS	\$0.10/1M reads	For application development and testing, or workloads with variable traffic.
M0	FREE	For learning and exploring MongoDB in a cloud environment.

En donde seleccionaremos el plan gratuito para realizar la exploración de la base de datos.



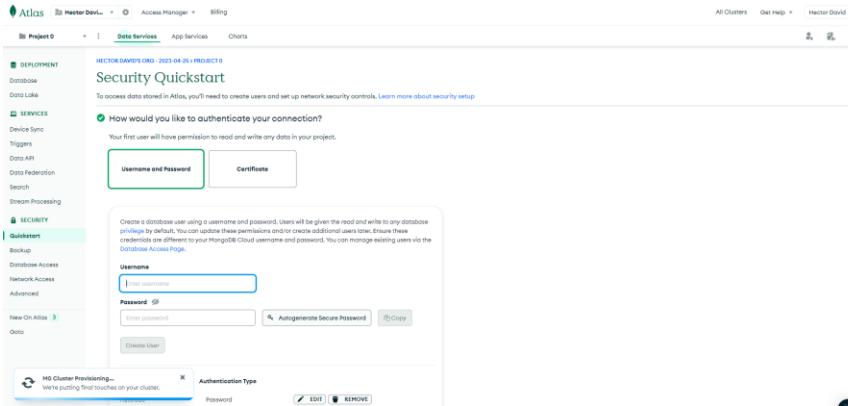
Dejamos tal cual la información que nos sale por default:

The screenshot shows the MongoDB Atlas cluster creation interface. It includes fields for Provider (aws), Region (N. Virginia (us-east-1)), Name (mern-web-personal), Tag (optional), and Auto-scale.

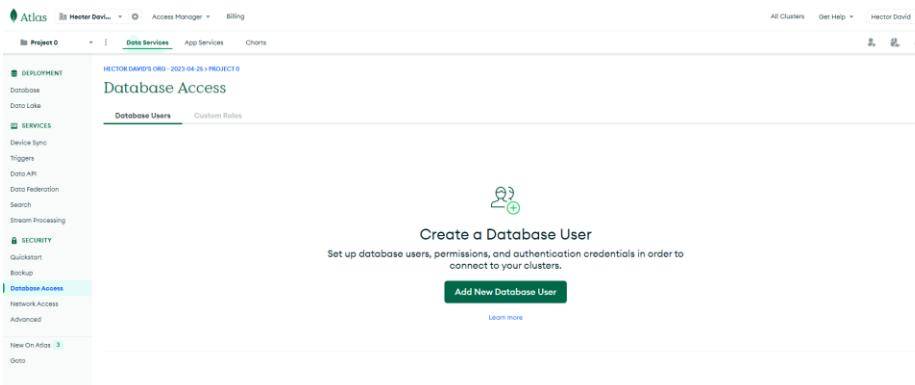
Acá colocamos el nombre de nuestro **cluster**, en mi caso lo dejare **mern-web-personal**



Y procedemos a darle en **create**, allí nos saldrá lo siguiente:



Ahora vamos a configurar el **database Access**



Acá vamos a crear un usuario y contraseña para poder acceder a nuestra **db**, en mi caso utilizare el usuario **mern** y colocare una contraseña, además de agregarle un rol de **admin**.



Add New Database User

Create a database user to grant an application or user access to databases and collections in your clusters in this Atlas project. Granular access control can be configured with default privileges or custom roles. You can grant access to an Atlas project or organization using the corresponding [Access Manager](#).

Authentication Method



MongoDB uses [SCRAM](#) as its default authentication method.

Password Authentication

password: mern
password: SHOW

Database User Privileges

Configure role based access control by assigning database user a mix of one built-in role, multiple custom roles, and multiple specific privileges. A user will gain access to all actions within the roles assigned to them, not just the actions those roles share in common. You must choose at least one role or privilege. [Learn more about roles](#).

Built-in Role SELECTED

Select one [built-in role](#) for this user.

Y procedemos a crearlo, ahora vamos a agregar la **ip** para acceder a la **db**, nos dirigimos al menú **network Access**.

Le damos en **add IP Address**

Seleccionamos **Allow Access From Anywhere**, en este caso recordemos que realizaremos conexiones para aprender, pero en un entorno de producción no lo colocaremos de esta manera.



De esta manera tendremos configurada nuestra **db** para poder iniciar con la carga de la información.

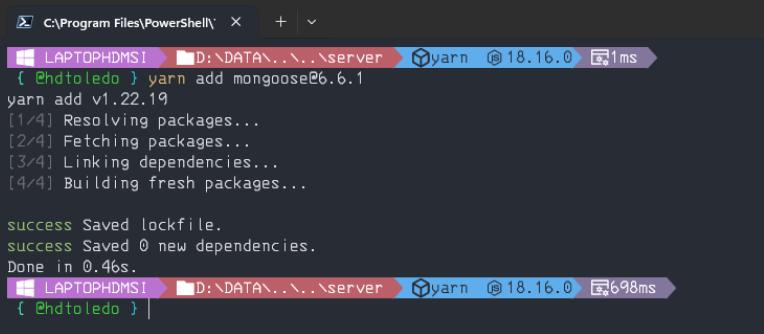
MONGOOSE – CONECTANDO LA APP CON LA DB

Para realizar nuestra conexión a la **DB** vamos a utilizar la dependencia de **mongoose**, nos dirigimos a la página de **yarn** y la buscamos

Ingresamos a **mongoose**, y en la página de **yarn** vamos a seleccionar la versión 6.6.1 allí nos dan las indicaciones de cómo debemos instalar



Para instalarlo vamos a nuestra terminal y vamos a colocar el comando de instalación **yarn add mongoose@6.6.1** y de esta manera ya tenemos instalado **mongoose** en nuestro server, verificamos en nuestro **package.json** que la dependencia este instalada:



```
package.json > ...
1  {
2    "name": "web-personal-api",
3    "version": "0.0.1",
4    "description": "API web Personal",
5    "main": "index.js",
6    "author": "@hdtoledo",
7    "license": "MIT",
8    "dependencies": {
9      "mongoose": "6.6.1"
10   }
11 }

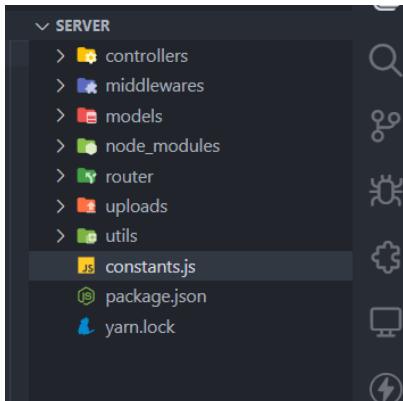
LAPTOPHDMI\> C:\Program Files\PowerShell\> D:\DATA\..\..\server> yarn v1.22.19
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 0 new dependencies.
Done in 0.46s.

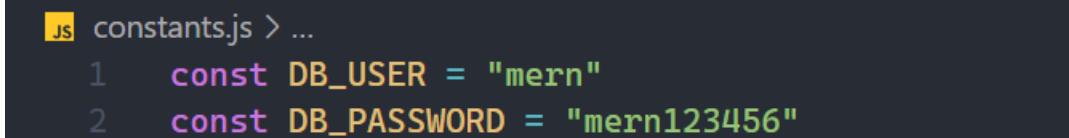
LAPTOPHDMI\> D:\DATA\..\..\server> yarn v1.22.19
yarn v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 0 new dependencies.
Done in 0.46s.
```

Ahora vamos a ir almacenando en un archivo de constantes las variables que vamos a utilizar en nuestro app, creamos nuestro archivo **constants.js** y colocamos lo siguiente:



Dentro de nuestro archivo asignaremos unas variables:



```
constants.js > ...
1 const DB_USER = "mern"
2 const DB_PASSWORD = "mern123456"
```

Y ahora nos dirigimos a nuestra página inicial de atlas en **mongodb** y le damos clic en **connect**



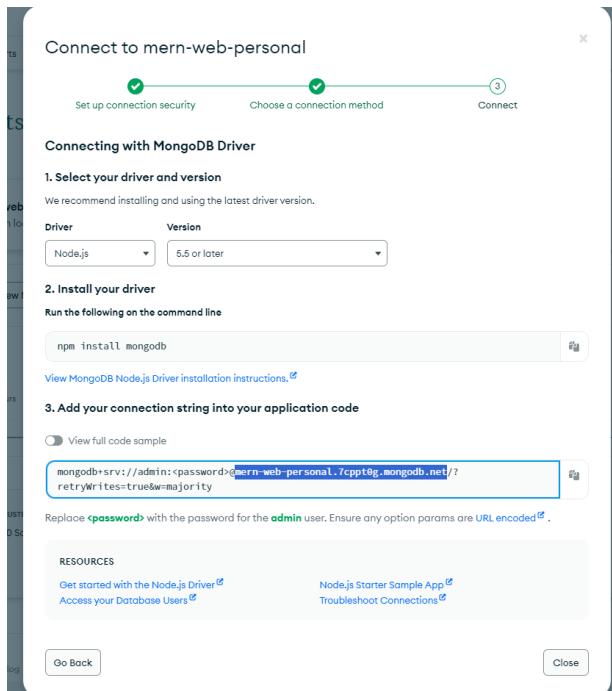
The screenshot shows the MongoDB Atlas interface for the 'mern-web-personal' database deployment. It includes a sidebar with options like Deployment, Services, Security, and New On Atlas. The main area displays monitoring data for connections and data size, and a button to load sample datasets.

Damos clic en **drivers**:

The screenshot shows a modal window titled 'Connect to mern-web-personal'. It has three tabs: 'Set up connection security', 'Choose a connection method', and 'Connect'. Under 'Choose a connection method', there is a section titled 'Connect to your application' which lists 'Drivers' as the selected option. Other options include 'Compass', 'Shell', 'MongoDB for VS Code', and 'Atlas SQL'.

Y allí vamos a seleccionar la dirección que tenemos seleccionada:





Esta dirección la vamos a agregar a una **const** de la siguiente manera:

```
JS constants.js > ...
1 const DB_USER = "mern"
2 const DB_PASSWORD = "mern123456"
3 const DB_HOST = "mern-web-personal.7cppt0g.mongodb.net"
4
```

Procedemos a crear dos **const** más en las cuales vamos a vincular la versión de nuestra API y la dirección del servidor:

```
JS constants.js > IP_SERVER
1 const DB_USER = "admin"
2 const DB_PASSWORD = "admin123456"
3 const DB_HOST = "mern-web-personal.7cppt0g.mongodb.net"
4
5 const API_VERSION = "v1"
6 const IP_SERVER = "localhost"
```

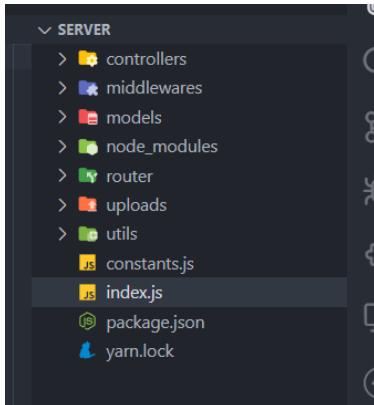
Ahora debemos exportar el módulo de las **const** de la siguiente manera:

```
JS constants.js > ...
1 const DB_USER = "admin"
2 const DB_PASSWORD = "admin123456"
3 const DB_HOST = "mern-web-personal.7cppt0g.mongodb.net"
4
5 const API_VERSION = "v1"
6 const IP_SERVER = "localhost"
7
8 module.exports = {
9   DB_USER,
10  DB_PASSWORD,
11  DB_HOST,
12  API_VERSION,
13  IP_SERVER
14 }
```



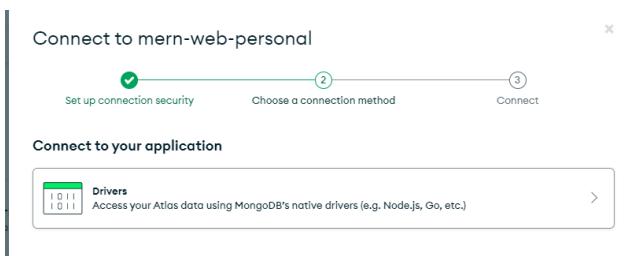
@hdtoledo

Ahora vamos a crear nuestro **index.js** y dentro de este vamos a traer **mongoose** a través del **cliente** y el **serverApi**, además, traemos las variables que vamos a usar de **constants.js**

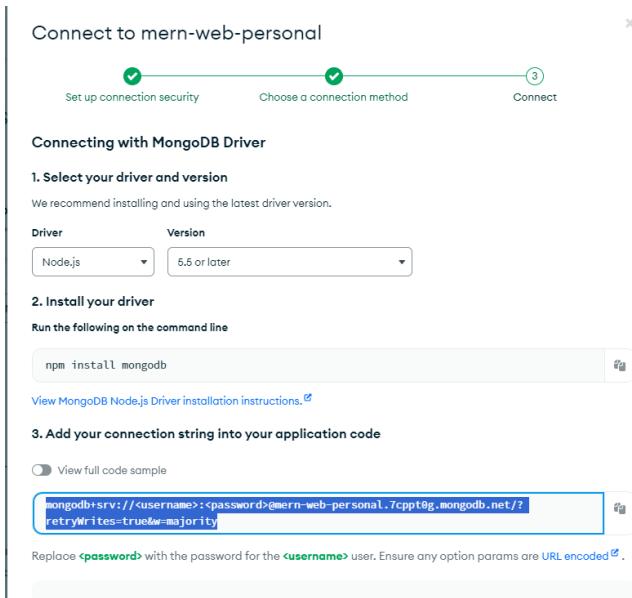


```
JS index.js > ...
1 const mongoose = require("mongoose")
2 const { DB_USER, DB_PASSWORD, DB_HOST, IP_SERVER, API_VERSION } = require("./constants.js")
3 |
```

Ahora procedemos a crear la conexión con **mongoose** y debemos copiar de atlas la dirección completa, al momento de darle **connect** en atlas seleccionamos **drivers**



Y allí nos vamos a la parte de abajo y seleccionamos:



Agregamos lo siguiente:

```
JS index.js > ...
1 const mongoose = require("mongoose")
2 const { DB_USER, DB_PASSWORD, DB_HOST, IP_SERVER, API_VERSION } = require("./constants.js")
3
4
5 mongoose.set('strictQuery', false)
6
7 mongoose.connect(
8   `mongodb+srv://${DB_USER}:${DB_PASSWORD}@${DB_HOST}/`,
9   (error) => {
10     if (error) throw error
11     console.log("La conexión al servidor ha sido exitosa!");
12   }
13 )
14
15 |
```

- **mongoose.set('strictQuery', false)**

Esta línea configura el valor de la opción strictQuery de Mongoose en false. Esto permite que Mongoose acepte consultas que no cumplen con las restricciones de validación predeterminadas. Esto puede ser útil para aplicaciones que necesitan compatibilidad con consultas antiguas o que necesitan realizar consultas que no se pueden validar fácilmente, en nuestro caso lo aplicamos ya que vamos a trabajar con una versión anterior a la actual.

- Dentro de esta conexión utilizamos los backticks, **mongoose.connect(`mongodb+srv://\${DB_USER}:\${DB_PASSWORD}@\${DB_HOST}/`, (error) => { if (error) throw error // ... })`**. Esta línea conecta a la base de datos MongoDB utilizando las credenciales especificadas en las variables de entorno `DB_USER`, `DB_PASSWORD` y `DB_HOST`. Si la conexión falla, se lanza un error. *

Ahora llego la hora de probar si funciona para ello vamos a modificar nuestro **package.json** y vamos a agregar el script **start**:

```
JSON package.json > ...
1 {
2   "name": "web-personal-api",
3   "version": "0.0.1",
4   "description": "API web Personal",
5   "main": "index.js",
6   "author": "@hdtoledo",
7   "license": "MIT",
8   "scripts": {
9     "start": "node index.js"
10   },
11   "dependencies": {
12     "mongoose": "6.6.1"
13   }
14 }
15 |
```

Y ahora procedemos a ejecutar la consola con el comando **yarn start**, si todo va bien nos mostrara el mensaje de que se conectó correctamente:



```
LAPTOPHDMI D:\DATA\...\server yarn 18.16.0 1ms
{ @hdtledo } yarn start
yarn run v1.22.19
$ node index.js
La conexión al servidor ha sido exitosa!
```

CREANDO EL SERVIDOR HTTP CON EXPRESS

Express.js es un marco de trabajo (framework) para construir aplicaciones web en Node.js. Piensa en **Express.js** como un conjunto de herramientas predefinidas que hacen que sea más fácil crear sitios web y aplicaciones web en el lenguaje de programación JavaScript.

Express.js simplifica tareas comunes en el desarrollo web, como manejar rutas, gestionar solicitudes y respuestas, trabajar con bases de datos y mucho más. Te permite construir aplicaciones web de manera más rápida y eficiente al proporcionarte una estructura y un conjunto de funciones que facilitan la creación de servidores web y la gestión de rutas y recursos.

Ahora procedemos a agregarlo a nuestro proyecto con el siguiente comando

```
LAPTOPHDMI D:\DATA\...\server yarn 18.16.0 178ms
{ @hdtledo } yarn add express@4.18.1
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 1 new dependency.
info Direct dependencies
└ express@4.18.1
info All dependencies
└ express@4.18.1
Done in 1.46s.
```

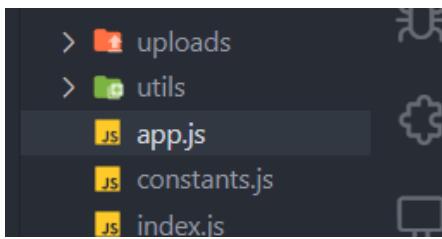
Una vez instalado verificamos en nuestro **package.json** que la dependencia haya sido instalada:

```
8  "scripts": {
9    "start": "node index.js"
10   },
11  "dependencies": {
12    "express": "4.18.1",
13    "mongoose": "6.6.1"
14  }
15 }
16 }
```



@hdtledo

Ahora procedemos a crear nuestro **app.js** en donde colocaremos todo lo relacionado a **express**



Dentro vamos a colocar lo siguiente y vamos a dejar unas tareas pendientes las cuales vamos a ir realizando paso a paso.

```
js app.js > ...
1 const express = require('express')
2 const { API_VERSION } = require("./constants")
3
4 const app = express()
5
6 // Importar rutas
7
8
9 //Configurar Body Parse
10
11
12 // Configurar Header HTTP - CORS
13
14 // Configurar Rutas
15
16
17 module.exports = app
```

Por el momento dejaremos allí la configuración y vamos a pasar a nuestro **index.js**, para realizar el levantamiento del servidor, hacemos la importación de **express** mediante **app**

```
js index.js > app
1 const mongoose = require("mongoose")
2 const app = require("./app.js")
3 const { DB_USER, DB_PASSWORD, DB_HOST, IP_SERVER, API_VERSION } = require ("./constants.js")
4
```

Ahora vamos a levantar el servicio configurando el puerto a través de una const:

```
js index.js > ...
1 const mongoose = require("mongoose")
2 const app = require("./app.js")
3 const { DB_USER, DB_PASSWORD, DB_HOST, IP_SERVER, API_VERSION } = require ("./constants.js")
4
5
6 const PORT = process.env.PORT || 3000
7
8 mongoose.set('strictQuery', false)
9
```



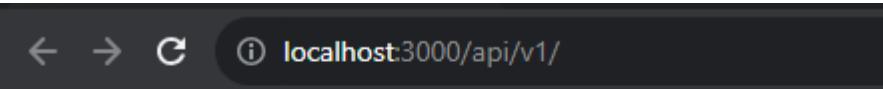
Y por último vamos a configurar la escucha de nuestro servidor en el puerto y que nos muestre un mensaje en consola:

```
42
13 mongoose.set("strictQuery", false);
14
15 mongoose.connect(
16   `mongodb+srv://${DB_USER}:${DB_PASSWORD}@${DB_HOST}/`,
17   (error) => {
18     if (error) throw error
19
20     app.listen(PORT, () => {
21       console.log(`#####
22       #### MERN API REST #####
23       #####`)
24       console.log(`http://:${IP_SERVER}:${PORT}/api/${API_VERSION}`)
25     })
26   }
27 )
28
```

Si todo va bien ejecutamos **yarn start** y nos devolverá lo siguiente:

```
LAPTOPHDMI ➜ D:\DATA\..\server ➜ yarn 18.16.0 2ms
{ hdtledo } yarn start
yarn run v1.22.19
$ node index.js
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
```

Ahora que tenemos levantado el servicio nos dirigimos al navegador a la dirección y el puerto y nos debe salir lo siguiente:



De esta manera ya tenemos nuestro servidor escuchando.



RECARGANDO EL SERVIDOR AUTOMATICAMENTE

Vamos a realizar un cambio sobre nuestros scripts de ejecución, y para ello vamos a añadir el script de desarrollo **DEV**, lo primero es hacer la instalación de la dependencia **nodemon** que nos va a permitir cargar un servicio sobre los cambios encontrados en nuestros recursos y que permitirá reiniciar el servidor para que los cambios se vean reflejados en nuestro navegador, ejecutamos en consola el comando **yarn add nodemon@2.0.20**

```
LAPTOPHDM5I D:\DATA\..\..\server yarn 18.16.0 1ms
( @hdtledo ) yarn add nodemon@2.0.20
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 1 new dependency.
info Direct dependencies
└─ nodemon@2.0.20
info All dependencies
└─ nodemon@2.0.20
Done in 1.06s.
```

Verificamos en nuestro **package.json** que este instalada:

```
10  },
11  "dependencies": {
12    "express": "4.18.1",
13    "mongoose": "6.6.1",
14    "nodemon": "2.0.20"
15  }
16 }
17 }
```

Ahora en nuestros scripts agregamos **dev**:

```
7   "license": "MIT",
8    ▷ Depurar
9   "scripts": {
10     "start": "node index.js",
11     "dev": "nodemon index.js"
12   },
13   "dependencies": {
```

Y ya solo nos queda ejecutar **yarn dev** en nuestra terminal para que se vea el resultado:

```
LAPTOPHDM5I D:\DATA\..\..\server yarn 18.16.0 1ms
( @hdtledo ) yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
```



@hdtledo

Vamos a realizar un cambio sobre nuestro **index.js** y una vez guardados los cambios se van a ver reflejados en la terminal y en el navegador:

The image shows a code editor on the left containing the **index.js** file. On the right is a terminal window titled "C:\Program Files\PowerShell\" showing the output of a **yarn run v1.22.19** command. The terminal output shows the execution of **node index.js**, the server starting at port 3000, and the URL **http://localhost:3000/api/v1**.

```
index.js > mongoose.connect() callback > app.listen() callback
  1 const mongoose = require("mongoose")
  2 const app = require("./app.js")
  3 const {
  4   DB_USER,
  5   DB_PASSWORD,
  6   DB_HOST,
  7   IP_SERVER,
  8   API_VERSION,
  9 } = require("./constants.js")
 10
 11 const PORT = process.env.PORT || 3000
 12
 13 mongoose.set("strictQuery", false)
 14
 15 mongoose.connect(
 16   `mongodb+srv://${DB_USER}:${DB_PASSWORD}@${DB_HOST}/`,
 17   (error) => {
 18     if (error) throw error
 19
 20     app.listen(PORT, () => {
 21       console.log("#####")
 22       console.log("## Probando ##")
 23       console.log("#####")
 24       console.log(`http://${IP_SERVER}:${PORT}/api/${API_VERSION}`)
 25     })
 26   }
 27 )
```

```
LAPTOPHDMI1 D:\DATA\..\server yarn 18.16.0 1ms
{ @hdtledo } yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
=====
### MERN API REST ###
=====
http://localhost:3000/api/v1
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
=====
### Probando ###
=====
http://localhost:3000/api/v1
```

Cada vez que hacemos cambios en nuestro código y los guardamos se verán reflejados y **nodemon** reiniciara el servicio para que nosotros lo podamos ver.

CONFIGURANDO BODY PARSER

El middleware **body-parser** se utiliza para analizar y extraer los datos del cuerpo (**body**) de una solicitud HTTP, en el contexto de una aplicación web, cuando un cliente envía datos al servidor a través de un formulario **HTML** o una solicitud **API**, esos datos se envían en el cuerpo de la solicitud **HTTP**. **body-parser** se encarga de tomar esos datos en formato de texto y convertirlos en un objeto JavaScript utilizable para que puedas trabajar con ellos en tu aplicación.

Por ejemplo, si estás construyendo un formulario de registro en una aplicación web, cuando un usuario complete el formulario y lo envíe, los datos ingresados (como el nombre, el correo electrónico y la contraseña) se enviarán en el cuerpo de la solicitud HTTP. **body-parser** ayudará a tu aplicación Express a tomar esos datos del cuerpo de la solicitud y convertirlos en un objeto JavaScript que puedes usar para registrar al usuario en tu base de datos, por ejemplo.

Así que, en resumen, **body-parser** es una herramienta esencial para procesar y gestionar datos enviados desde el cliente a través de solicitudes HTTP en una aplicación **Express.js**, procedemos a realizar la instalación con el comando **yarn add body-parser@1.20.0**.

The image shows a terminal window titled "LAPTOPHDMI1" with the command **yarn add body-parser@1.20.0** being run. The output shows the progress of the installation, including resolving packages, fetching dependencies, linking them, and building fresh packages. It ends with a success message and a duration of 0.47s.

```
LAPTOPHDMI1 D:\DATA\..\server yarn 18.16.0 1ms
{ @hdtledo } yarn add body-parser@1.20.0
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 0 new dependencies.
Done in 0.47s.
```



@hdtledo

Verificamos en nuestro **package.json** que este instalado:

```
12  "dependencies": {  
13    "body-parser": "1.20.0",  
14    "express": "4.18.1",  
15    "mongoose": "6.6.1",  
16    "nodemon": "2.0.20"  
17  }  
18 }
```

Ahora vamos a configurar nuestro **app.js** con el **body parser**:

```
js app.js > ...  
1  const express = require("express")  
2  const bodyParser = require("body-parser")  
3  const { API_VERSION } = require("./constants")  
4  
5  const app = express()  
6  
7  // Importar rutas  
8  
10 //Configurar Body Parse  
11 app.use(bodyParser.urlencoded({ extended: true}))  
12 app.use(bodyParser.json())  
13  
14 // Configurar Header HTTP - CORS  
15  
16 // Configurar Rutas  
17  
18  
19 module.exports = app
```

Llamamos nuestro **body parser** a través de unas **const**, y configuramos para que todo lo que nos llegue a través de la **url** de nuestra **app** se convierta en **json**.



@hdtoledo

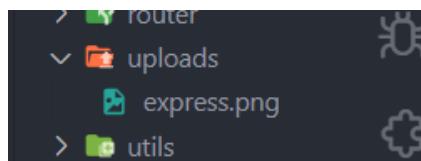
CONFIGURANDO CARPETA ESTATICOS

Todos los archivos **estáticos** los vamos a ubicar en la carpeta **uploads** en la cual tendremos imágenes, archivos, y demás cosas que necesitemos dentro de nuestra **app**, así que vamos a configurar la ubicación de esta carpeta dentro de nuestro archivo **app.js**:

```
js app.js > ...
1 const express = require("express")
2 const bodyParser = require("body-parser")
3 const { API_VERSION } = require("./constants")
4
5
6 const app = express()
7
8 // Importar rutas
9
10
11 //Configurar Body Parse
12 app.use(bodyParser.urlencoded({ extended: true}))
13 app.use(bodyParser.json())
14
15 //Configurar carpeta static
16 app.use(express.static("uploads"))
17
18 // Configurar Header HTTP - CORS
19
20 // Configurar Rutas
21
22
23 module.exports = app
```

Para comprobar que está funcionando es muy sencillo, guardaremos una imagen de prueba dentro de la carpeta **uploads** y simplemente levantamos nuestro servidor, una vez levantado nos dirigimos a nuestro navegador y en la raíz de nuestro **localhost**, añadimos **/imagen.jpg**, si todo va bien debemos obtener una vista de nuestra imagen.

Subimos la imagen:



Levantamos el servidor:

```
LAPTOPHDM! D:\DATA\...\server yarn 18.16.0 1ms
{ hdtoledo } yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
```



Revisamos en el navegador con la url de nuestra imagen:



De esta manera verificamos que este correctamente configurada nuestra carpeta de estáticos.

CONFIGURANDO LAS CORS

Las **CORS** (Cross-Origin Resource Sharing) son una medida de seguridad implementada en los navegadores web para prevenir ataques de seguridad relacionados con solicitudes entre diferentes dominios (sitios web). Básicamente, las **CORS** son un conjunto de reglas que dictan si un sitio web (dominio) tiene permiso para acceder a los recursos de otro sitio web (dominio).

Cuando estás construyendo una aplicación web con Express.js y deseas que tu servidor permita que otros dominios hagan solicitudes **HTTP** a tus rutas y recursos (como **APIs**), es importante configurar las **CORS** adecuadamente. Sin la configuración adecuada de las **CORS**, los navegadores web bloquearán las solicitudes desde otros dominios por razones de seguridad.

En **Express.js**, puedes habilitar las **CORS** utilizando middleware, como **cors**, que es una biblioteca de middleware específicamente diseñada para gestionar las políticas de **CORS**. Cuando configuras **cors** en tu aplicación **Express**, puedes especificar desde qué dominios permites solicitudes y qué tipos de solicitudes (como **GET**, **POST**, etc.) están permitidas.

Por ejemplo, puedes configurar **cors** para permitir que cualquier sitio web acceda a tus recursos o especificar dominios específicos que tienen permiso. También puedes definir las cabeceras que se deben incluir en las respuestas para indicar que las solicitudes **CORS** son permitidas.

Ahora vamos a configurar las **CORS** dentro de nuestro archivo **app.js**, para ello colocamos el comando **yarn add cors@2.8.5**

```
LAPTOPHDMI1 D:\DATA\..\..\server yarn 18.16.0 1ms
{ @hdtledo } yarn add cors@2.8.5
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 1 new dependency.
info Direct dependencies
└ cors@2.8.5
info All dependencies
└ cors@2.8.5
Done in 1.01s.
```

y ahora configuraremos de la siguiente manera nuestro **app.js** agregando **const cors**:

```
app.js > cors
1 const express = require("express")
2 const bodyParser = require("body-parser")
3 const cors = require("cors")
4 const { API_VERSION } = require("./constants")
5
```

Y haciendo la implementación de este con **app.use**

```
18
19 // Configurar Header HTTP - CORS
20 app.use(cors())
21
22 // Configurar Rutas
```

De esta manera dejamos las **cors** configuradas y si iniciamos nuestro servidor no nos debe generar ningún error.

```
LAPTOPHDMI1 D:\DATA\..\..\server yarn 18.16.0 1ms
{ @hdtledo } yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
```



CREANDO NUESTRO PRIMER ENDPOINT

Un "**endpoint**" se refiere a un punto de acceso o una URL específica a la que se puede hacer una solicitud HTTP para interactuar con una aplicación o un servicio. Los **endpoints** son como puertas de entrada a una aplicación o servicio web que te permiten realizar diversas operaciones.

Cada **endpoint** generalmente está asociado con una función o acción específica en el servidor, y suelen corresponder a operaciones CRUD (Crear, Leer, Actualizar y Borrar) en una base de datos o a la ejecución de una función específica en el servidor.

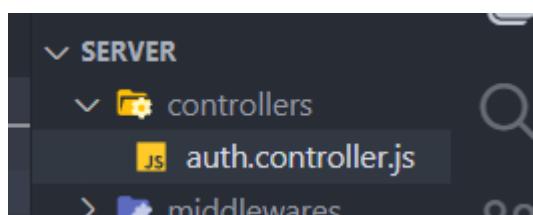
Por ejemplo, en una API de redes sociales, podrías tener los siguientes **endpoints**:

1. **/posts**: Este **endpoint** podría usarse para obtener una lista de publicaciones.
2. **/posts/{id}**: Usado para obtener una publicación específica por su ID.
3. **/users/{username}/posts**: Podría proporcionar las publicaciones de un usuario en particular.
4. **/posts/create**: Para crear una nueva publicación.
5. **/posts/{id}/update**: Para actualizar una publicación existente.
6. **/posts/{id}/delete**: Para eliminar una publicación.

Cada uno de estos **endpoints** puede aceptar diferentes tipos de solicitudes HTTP, como GET para obtener datos, POST para crear datos, PUT para actualizar datos y DELETE para eliminar datos, entre otros métodos HTTP.

Ahora empezaremos a realizar el montaje de los **endpoints** para nuestros usuarios y vamos a validar el sistema de autenticación, registro de usuario y login, tambien vamos a configurar el refresh token del usuario.

Para iniciar vamos a crear nuestro controlador de **auth.controller.js**, para ello nos ubicamos en la carpeta **controllers** y creamos el archivo

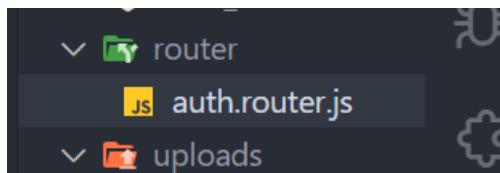


Dentro de nuestro **auth** colocamos lo siguiente:

```
1 function register(req, res){  
2     console.log("Se ha ejecutado el registro")  
3  
4     res.status(200).send({msg:"Funciono Perfecto !"})  
5 }  
6  
7 module.exports = {  
8     register,  
9 }
```

Acá podemos observar como a través de una función pasamos un request y response y mediante consola por el momento le vamos a indicar que se ha ejecutado el registro, también agregamos un mensaje de estado 200 para indicar que todo va bien y por último exportamos el módulo para poderlo utilizar.

Ahora la pregunta es cómo hacemos para que funcione nuestro controlador, y para ello vamos a crear nuestra ruta, en la carpeta **router** vamos a crear nuestra ruta creamos el archivo **auth.router.js**



Ahora vamos a colocar lo siguiente para empezar a utilizar nuestra autenticación en **auth.router.js**:

```
router > js auth.router.js > AuthController
1 const express = require("express")
2 const AuthController = require("../controllers/auth.controller")
3
4 const api = express.Router()
5
6 api.post("/auth/register/", AuthController.register)
7
8 module.exports = api
```

De esta manera le indicamos que vamos a utilizar express y que vamos a utilizar nuestro controlador a través de la ruta **/auth/register/** por último exportamos el módulo.

Ahora para que nos funcione debemos pasarlo a través de **app.js** y hacer la importación para que express lo pueda utilizar:

```
8
9 // Importar rutas
10 const authRoutes = require("./router/auth.router")
11
12 //Configurar Body Parse
```

Y hacemos la configuración en el mismo **app.js** en donde dejamos el comentario de configurar rutas:

```
21
22 // Configurar Rutas
23 app.use(`/api/${API_VERSION}`, authRoutes)
24
```

De esta manera tenemos ya configurada nuestras rutas para poder ingresar, ahora debemos probar esta ruta y para ello vamos a utilizar **insomnia**, es una aplicación o software de código abierto diseñado para ayudar a los desarrolladores a probar y depurar **APIs** (Interfaces de Programación de Aplicaciones) de manera eficiente. Es una herramienta de cliente **REST** y **GraphQL** que permite a los desarrolladores interactuar con **API endpoints**, enviar solicitudes **HTTP**, ver respuestas y analizar los resultados.

Las principales características de Insomnia incluyen:

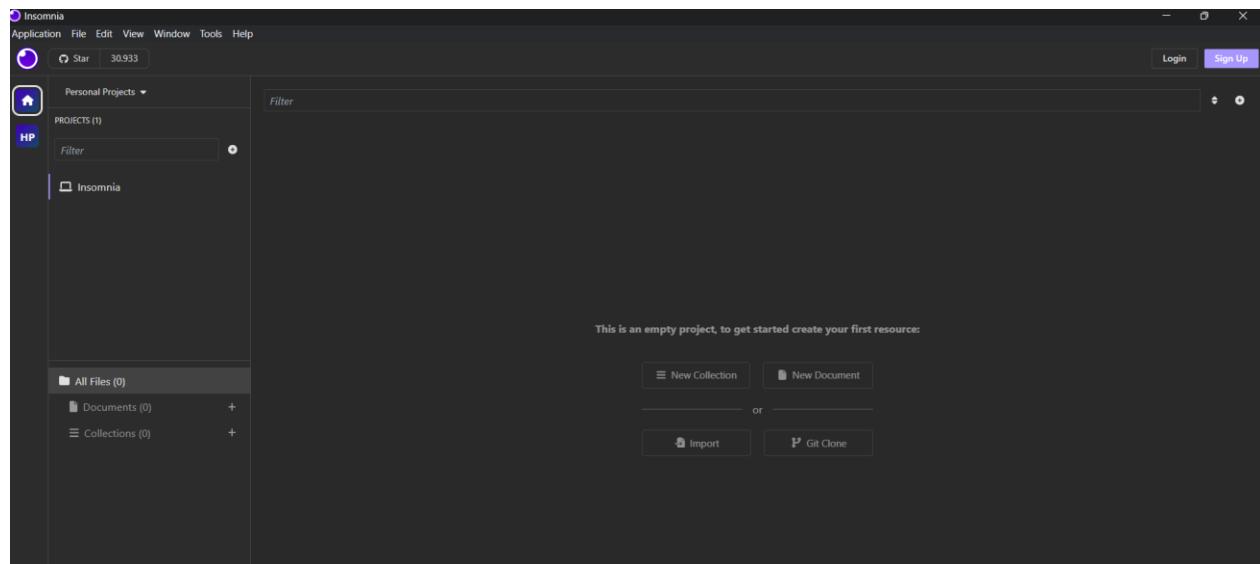
1. Interfaz Gráfica Amigable: Insomnia proporciona una interfaz de usuario intuitiva que facilita la creación, modificación y organización de solicitudes HTTP. Esto es especialmente útil para



aquellos que prefieren una interfaz visual en lugar de trabajar directamente con herramientas de línea de comandos.

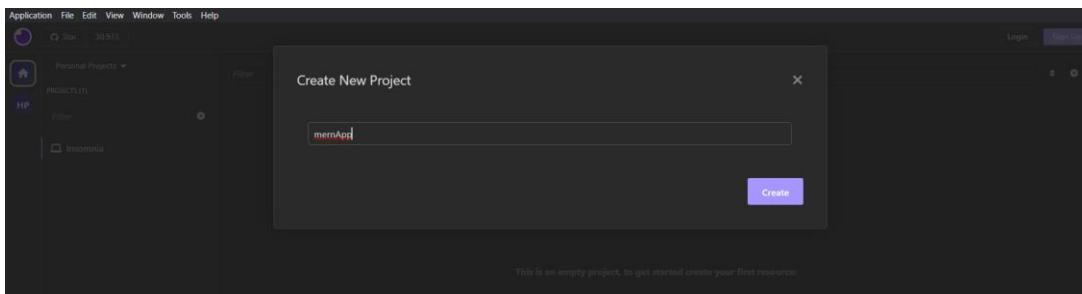
2. Sesiones de Trabajo: Permite organizar solicitudes y respuestas en sesiones de trabajo para proyectos específicos o para diferentes partes de una API. Esto ayuda a mantener todo ordenado y fácil de gestionar.
3. Soporte para Múltiples Protocolos: Además de HTTP, Insomnia admite otros protocolos como GraphQL, WebSockets, gRPC y más. Esto es útil cuando se trabaja con diferentes tipos de API.
4. Variables y Entornos: Insomnia permite definir variables y entornos, lo que facilita la personalización de las solicitudes para diferentes escenarios o ambientes.
5. Autenticación: Proporciona opciones para configurar fácilmente la autenticación en las solicitudes, como agregar encabezados de autorización o tokens.
6. Generación de Código: Puede generar código en varios lenguajes de programación a partir de solicitudes, lo que facilita la integración de las API en aplicaciones.
7. Documentación Interactiva: **Insomnia** permite crear documentación interactiva de API para compartir con otros miembros del equipo o con terceros.

Ahora procedemos a abrir **insomnia** y vamos a realizar una pequeña configuración, la apariencia que veremos será la siguiente:

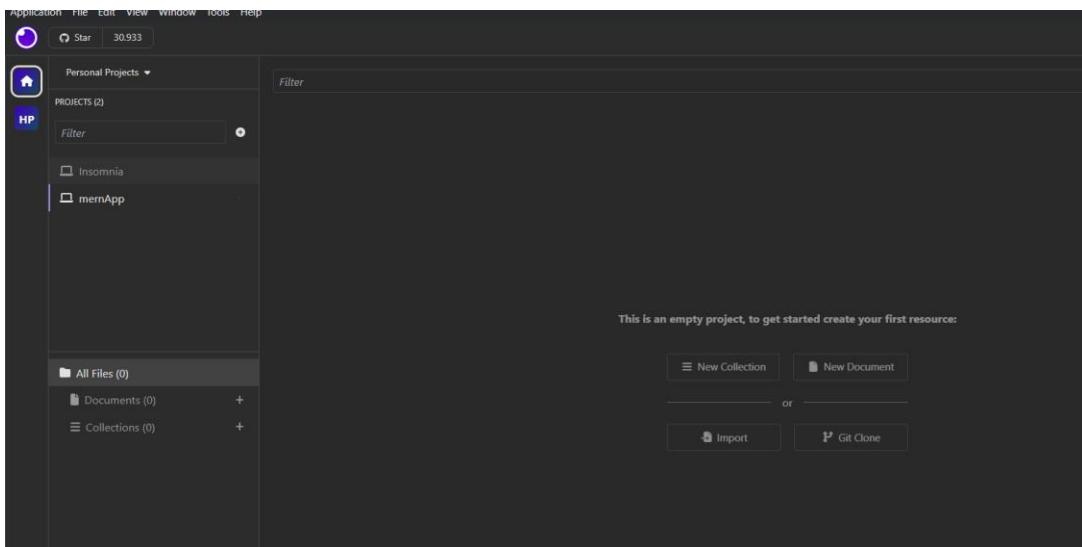


@hdtoledo

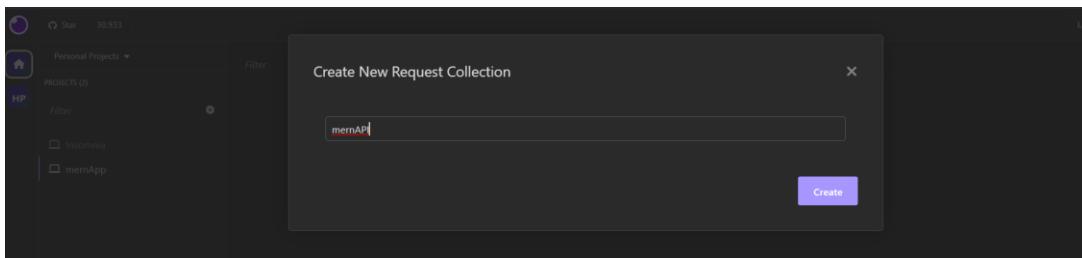
Ahora pasaremos a darle a crear un nuevo proyecto:



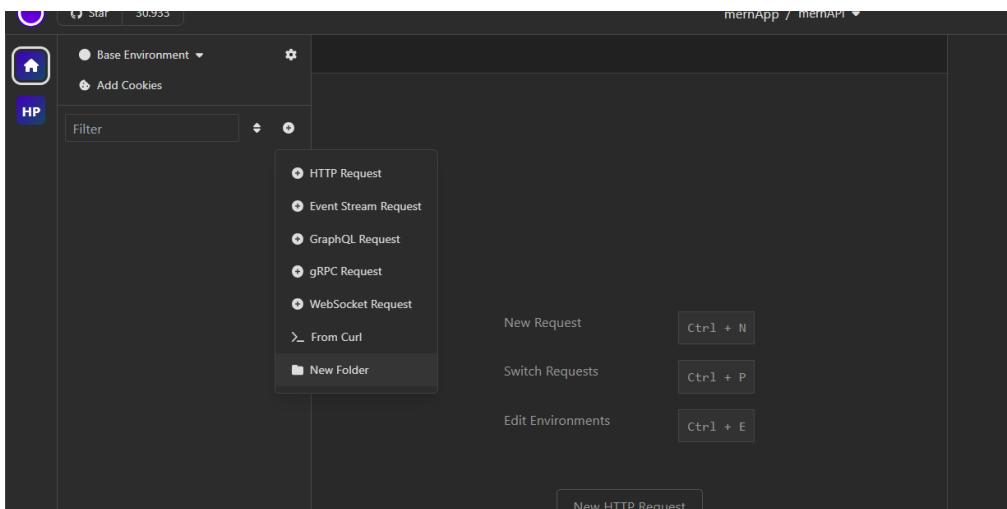
En nuestra interfaz nos mostrara el proyecto nuevo y vamos a crear una nueva colección:



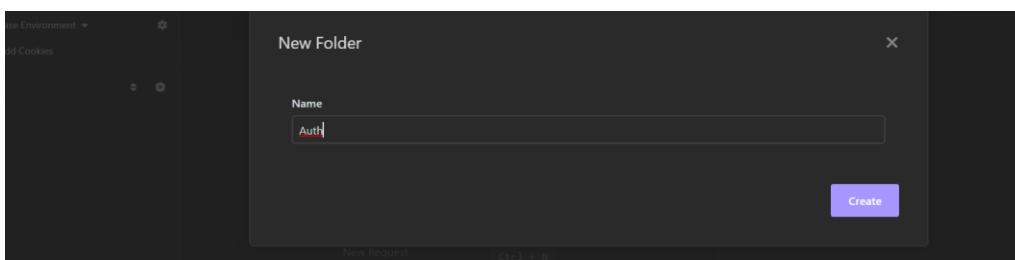
Creamos la nueva colección



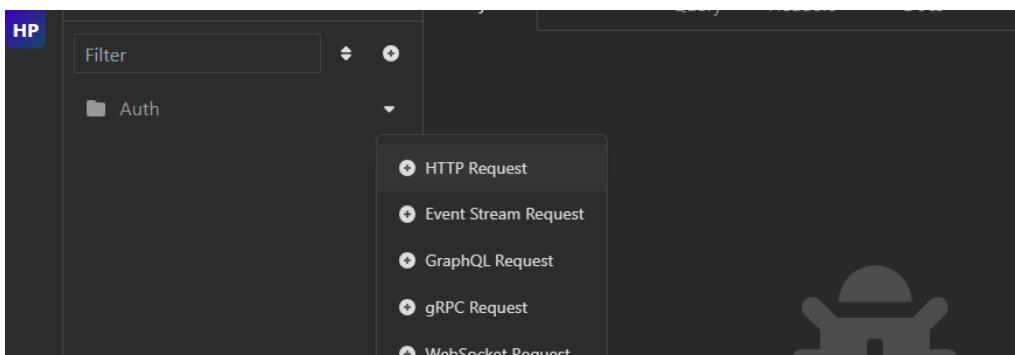
Y dentro de la interfaz nos mostrara para agregar unas variables de entorno, opción para cookies y luego tenemos una opción para agregar una nueva carpeta, presionamos para crear una nueva carpeta:



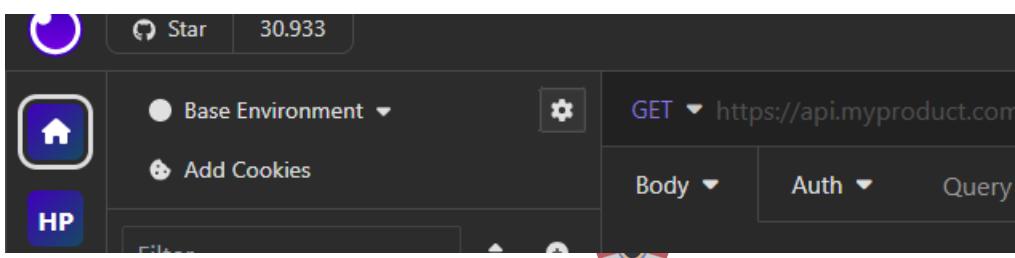
A la cual le colocamos Auth:



Y ahora le daremos de nuevo para crear una solicitud HTTP:

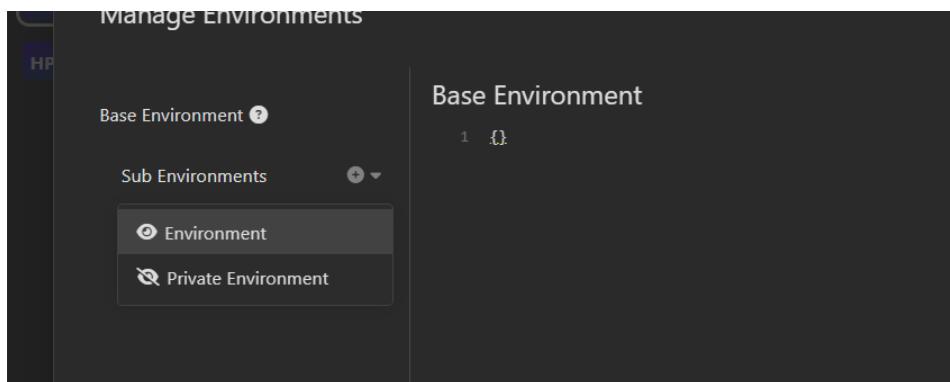


En la parte de arriba podemos observar donde dice **base environment** y le vamos a dar clic sobre la rueda dentada:

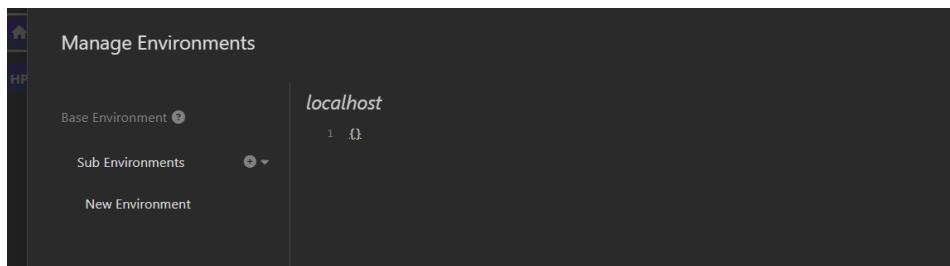


@hdtoledo

Y acá nos muestra para poder agregar unas variables de entorno, en este caso vamos a crear una de nuestra ruta principal a la cual estaremos haciendo las peticiones



Agregamos el nombre principal como lo es localhost:



Y dentro de las llaves vamos a colocar el **base path**



Recordemos que nuestro base path será la dirección principal de nuestro servidor:

```
HTTP/1.1 200 OK
#####
http://localhost:3000/api/v1
```

La seleccionamos, copiamos y pegamos en insomnia, y acá tenemos una opción para poder colocar colores a nuestros **paths** y distinguirlos en mi caso seleccionare un tono azul:

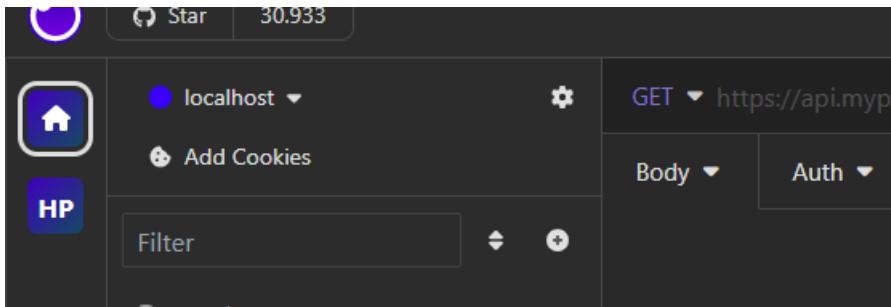


Y de esta manera nos debe quedar nuestro **base path**

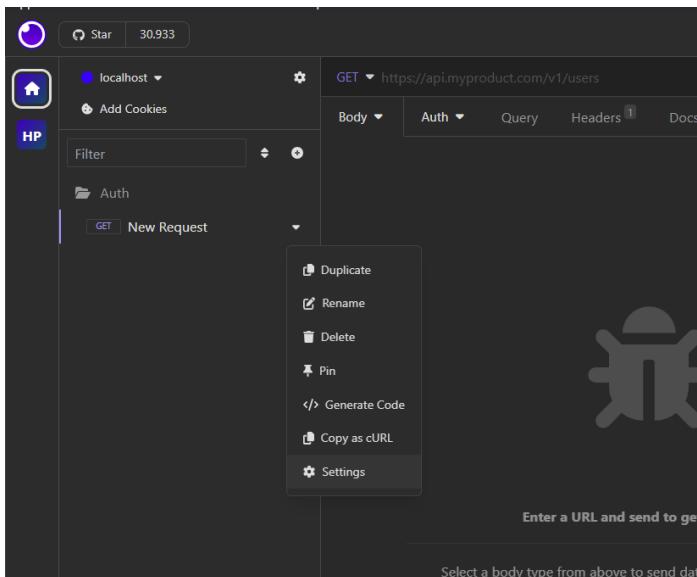


```
1: {
2:   "BASE_PATH": "http://localhost:3000/api/v1"
3: }
```

Cerramos y nos debe quedar así en nuestra vista:

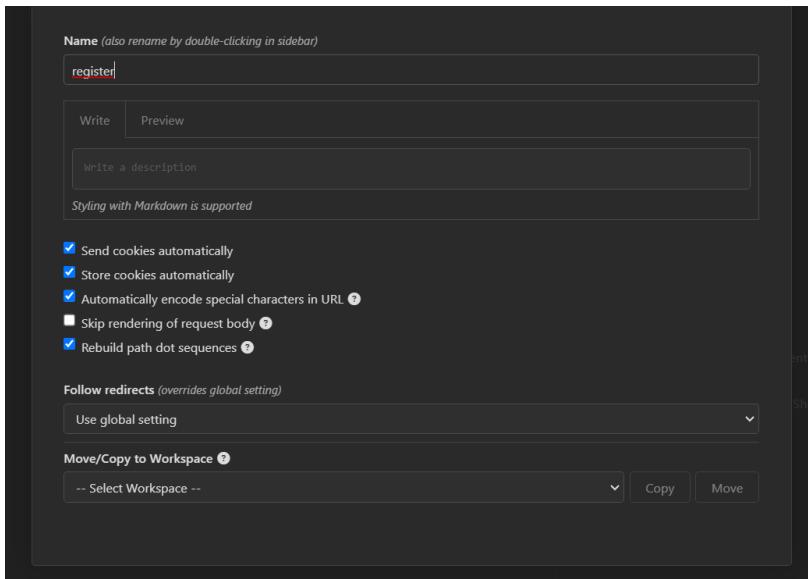


Ahora vamos a colocarle nombre a nuestra petición para ello le damos clic sobre las opciones de new request y seleccionamos settings

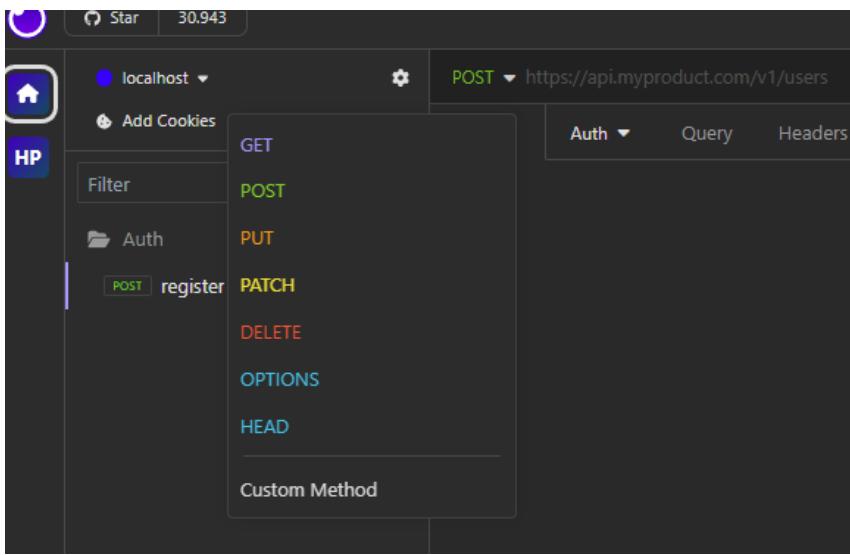


@hdtoledo

Y le vamos a colocar register



Al volver a nuestra vista anterior observamos como podemos hacer las diferentes peticiones HTTP:



Vamos a seleccionar el método **GET** para nuestra solicitud y al darle **ctrl + espacio** nos saldrá nuestro **base path** que configuramos:



Recordemos que nuestro base path es la ruta a la cual vamos a direccionar las solicitudes, en este caso la ruta es **http://localhost:3000/api/v1** de esta manera no tendremos que memorizar o estar escribiendo toda esta ruta, sino que simplemente la tendremos almacenada para poder agilizar el proceso.

Una vez seleccionada nuestra **base path** agregamos el resto de la ruta, que es **/auth/register** al hacer la petición de GET nos debe salir nuestro mensaje previamente configurado:

Y si revisamos nuestra terminal debe salir que se ha ejecutado correctamente el registro:

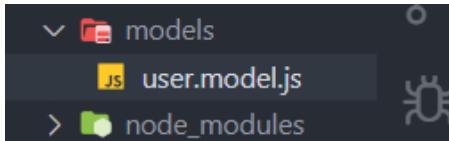
```
#####
#####
#####
#### API REST MERN ####
#####
http://localhost:3000/api/v1/
Se ha ejecutado el registro
```

De esta manera hemos configurado nuestro primer **endpoint**.



MODELO DEL USUARIO

El modelo del usuario es lo que tiene el usuario para poder iniciar su sesión, es decir nombres, apellidos, correo, contraseña, rol, activo, avatar, todas ellas, para ello nos vamos a **models** y vamos a crear el modelo **user.model.js**



Y dentro de nuestro **user.model.js**:

```
models > user.js > <unknown>
1  const mongoose = require("mongoose")
2
3  const UserSchema = mongoose.Schema({
4      firstname: {
5          type: String,
6          required: true,
7          trim: true
8      },
9      lastname: {
10         type: String,
11         required: true,
12         trim: true
13     },
14     email: {
15         type: String,
16         unique: true,
17         required: true
18     },
19     password: {
20         type: String,
21         required: true
22     },
23     role: String,
24     active: Boolean,
25     avatar: String,
26 })
27
28 module.exports = mongoose.model("User", UserSchema)
```

Primero hacemos la importación de **mongoose**, y asignamos en una **const** a través del esquema nuestro modelo de datos que vamos a pasarle, aquí tenemos nombres, apellidos, correo, contraseña, rol, activo, avatar, con tipos de datos **string**, **booleano**, además utilizamos para que sea requerido y que nos limpie los espacios en blanco que el usuario llegue a colocar utilizando **trim**.

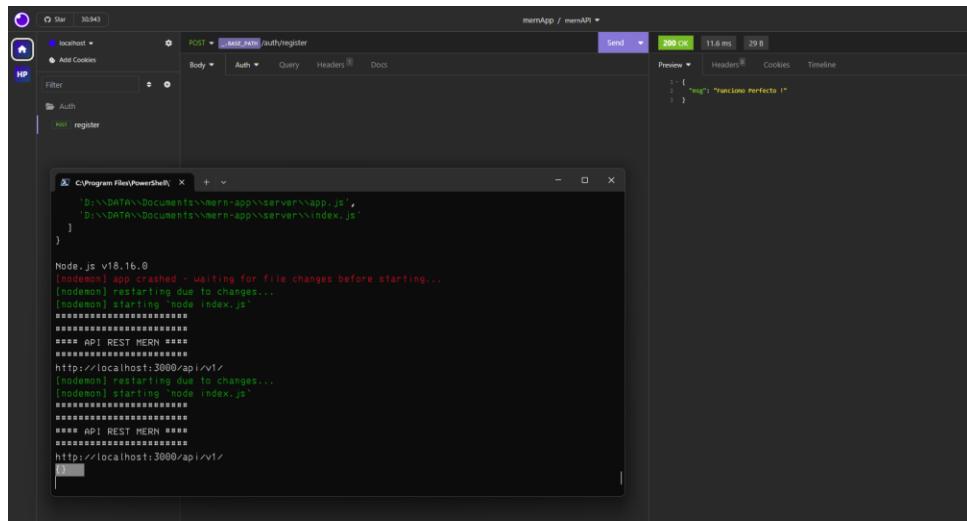


REGISTRANDO EL USUARIO

Para poder realizar el registro del usuario nos vamos a ubicar en nuestro controlador **auth.controller.js** y realizamos lo siguiente:

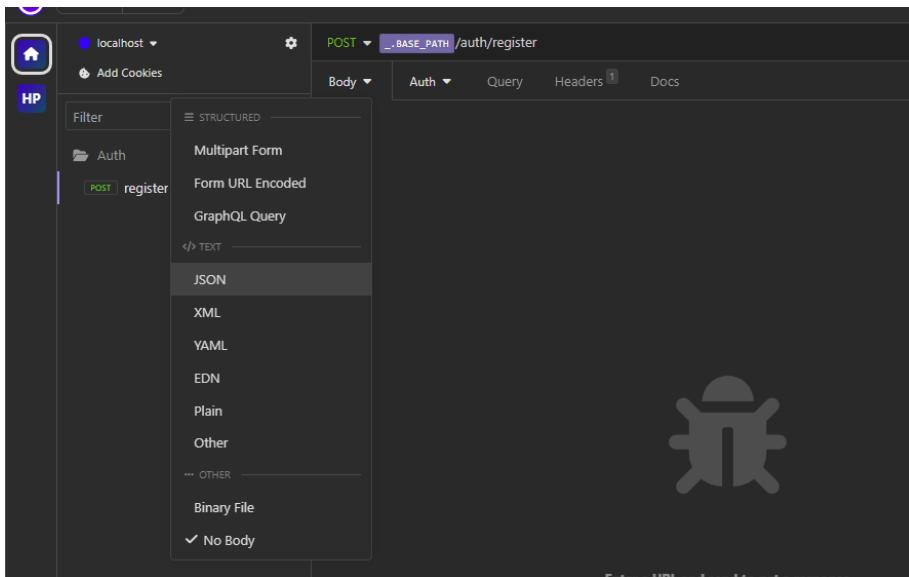
```
1 const User = require("../models/user")
2
3 function register(req, res){
4     console.log(req.body)
5
6     res.status(200).send({msg:"Funciono Perfecto !"})
7 }
8
9 module.exports = {
10     register,
11 }
```

A través de una **const** llamada **User** importamos nuestro **modelo**, y modificamos nuestro **console.log** para que nos capture lo que vamos a enviar en nuestro **body**, vamos a realizar el ejercicio para verificar que estamos enviando, nos vamos a insomnia y hacemos de nuevo la petición y vamos a obtener lo siguiente:

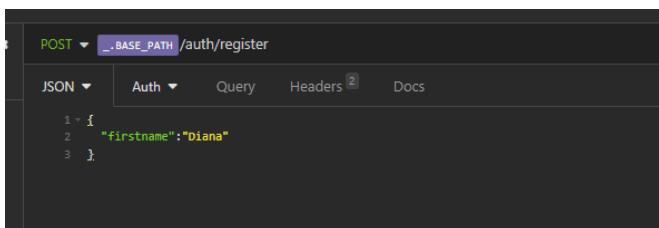


El mensaje se nos muestra de nuevo que todo va bien y en nuestra consola revisamos que hay un objeto vacío, esto quiere decir que todo va bien hasta el momento, lo siguiente es pasarle los datos a través de **JSON** y para ello en nuestro insomnia seleccionamos debajo del **base path** el formato de **JSON**:





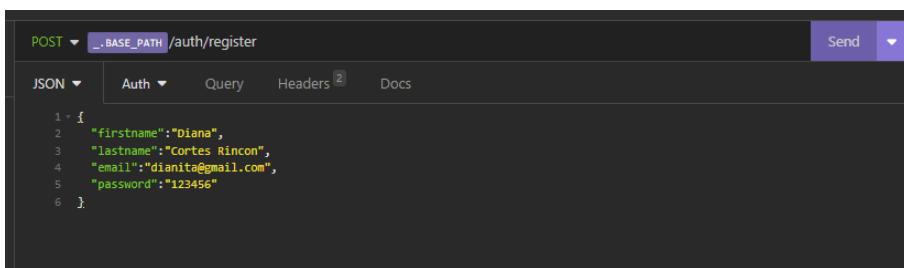
Una vez seleccionado JSON vamos a pasar los siguientes datos para probar:



Si volvemos a ejecutar la solicitud nos saldrá en nuestra consola los datos:

```
http://localhost:3000/api/v1/
{ firstname: 'Diana' }
```

Y ahora vamos a terminar de estructurar los datos que debemos pasar para verificar:



Estos datos serían los básicos al momento de nosotros realizar un registro de usuario, los demás datos como lo son rol, active y avatar los precargaremos luego, y ahora comprobamos de nuevo que se estén enviando estos datos:



The screenshot shows a browser's developer tools Network tab with a POST request to `/auth/register`. The request body is a JSON object:

```

1 + {
2   "firstname": "Diana",
3   "lastname": "Cortes Rincon",
4   "email": "dianita@gmail.com",
5   "password": "123456"
6 }

```

The response is a 200 OK status with a message: "Funciono Perfecto !".

Below the browser is a terminal window titled "push D:\DATA\Documents\mern-app\server" showing the command `yarn run v1.22.19` and the output of the Node.js application. It shows the application listening on port 3000 and receiving a POST request at `http://localhost:3000/api/v1/auth/register` with the same JSON data.

Ahora si vamos a realizar el registro verdadero, una vez comprobado que los datos se están enviando vamos a realizar lo siguiente en nuestro controlador:

```

controllers > ls auth.js > ...
1  const User = require("../models/user")
2
3  function register(req, res){
4    const {firstname, lastname, email, password} = req.body
5
6    if(!email) res.status(400).send({msg: "El Email es Obligatorio"})
7    if(!password) res.status(400).send({msg: "La Contraseña es Obligatoria"})
8
9    res.status(200).send({msg: "Funciono Perfecto !"})
10
11 module.exports = {
12   register,
13 }

```

Agregamos en una **const** los datos que vamos a tomar de nuestro **req.body** y hacemos unas condiciones para comprobar que se deben colocar los datos, revisamos de nuevo con **insomnia** y hacemos un pequeño error con el nombre de email y nos debe de salir así:

The screenshot shows a POST request to `/auth/register` with an invalid JSON body:

```

1 + {
2   "firstname": "Diana",
3   "lastname": "Cortes Rincon",
4   "email2": "dianita@gmail.com",
5   "password": "123456"
6 }

```

The response is a 400 Bad Request status with the message: "El Email es Obligatorio".

Ahora coloquemos de nuevo nuestro `console.log`:



```

1 const User = require("../models/user")
2
3 function register(req, res){
4   const {firstname, lastname, email, password} = req.body
5   console.log(req.body)
6
7   if(!email) res.status(400).send({msg: "El Email es Obligatorio"})
8   if(!password) res.status(400).send({msg: "La Contraseña es Obligatoria"})
9
10  res.status(200).send({msg:"Funciona Perfecto !"})
11}
12
13 module.exports = {
14   register,
15 }

```

Hablemos un poco del tema de seguridad, nuestra contraseña es visible y no debemos pasar este dato de esta manera ya que la contraseña debe ir encriptada por temas de seguridad, así que vamos a configurar primero la creación del usuario y a pasar los datos como un objeto y allí hacemos la configuración de seguridad, realizamos lo siguiente en nuestro **auth.controller.js**:

```

1 const User = require("../models/user")
2
3 function register(req, res){
4   const {firstname, lastname, email, password} = req.body
5   console.log(req.body)
6
7   if(!email) res.status(400).send({msg: "El Email es Obligatorio"})
8   if(!password) res.status(400).send({msg: "La Contraseña es Obligatoria"})
9
10  const user = new User({
11    firstname,
12    lastname,
13    password,
14    email: email.toLowerCase(),
15    role: "User",
16    active: false,
17  })
18
19  console.log(user)
20
21  res.status(200).send({msg:"Funciona Perfecto !"})
22}
23
24 module.exports = {
25   register,
26 }

```

Pasamos a través de un objeto los datos de nuestro esquema, en el ámbito del rol vamos a dejar que los administradores se creen solamente a través de nuestro y para los demás serán usuarios normales, volvemos a ejecutar en insomnia la ejecución de los datos y como podemos observar nos saldrá lo siguiente:



```
[nodemon] starting `node index.js`
#####
##### API REST MERN #####
#####
http://localhost:3000/api/v1/
{
  firstname: 'Diana',
  lastname: 'Cortes Rincon',
  email: 'dianita@gmail.com',
  password: '123456',
  role: 'User',
  active: false,
  _id: new ObjectId("65158d6d095108fefafed6463")
}
```

Al ver los datos lo primero que nos fijamos es que la **contraseña** esta en texto plano y esto es un problema de seguridad, para ello vamos a encriptarla y vamos a utilizar **bcryptjs** una dependencia que nos permitirá realizar la encriptación, nos dirijimos a las dependencias de yarn:

The screenshot shows the yarn package manager's search results for 'bcryptjs'. On the left, there's a sidebar with navigation links like 'Get Started', 'Features', 'CLI', 'Configuration', 'Advanced', 'Discord', 'GitHub', and a search bar. The main area displays the 'bcryptjs' package details. It shows the package version is 2.4.3, released 7 years ago. A note says 'Types are available via @types/bcryptjs'. Below this is the 'README' section, which includes a 'bcrypt.js' file description, build status (Build static v0.4.3), and download statistics (7.7M downloads). The 'Security considerations' section is also visible.

Y vamos a colocar lo siguiente en consola **yarn add bcryptjs@2.4.3**

```
LAPTOPHDMI ~ D:\DATA\...\server yarn 18.16.0 14.09s
( Endtoledo ) yarn add bcryptjs
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
success Saved 1 new dependency.
info Direct dependencies
└─ bcryptjs@2.4.3
info All dependencies
└─ bcryptjs@2.4.3
Done in 1.41s.
```

Lo primero que realizaremos será la importación de **bcrypt**, dentro de nuestro **auth.controller.js**:

```
controllers > auth.controller.js > register
1  const bcrypt = require("bcryptjs")
2  const User = require("../models/user.model")
```



Y ahora lo que haremos será quitar la contraseña de nuestros objetos de datos y vamos a crear una **const** que se llamará **salt** y otra **hashPassword** en donde utilizaremos **bcrypt**:

```
11 const user = new User({
12   firstname,
13   lastname,
14   email: email.toLowerCase(),
15   role: "User",
16   active: false,
17 }
18
19 const salt = bcrypt.genSaltSync(10)
20 const hashPassword = bcrypt.hashSync(password, salt)
21
22 console.log(password)
23 console.log(hashPassword)
24
25 res.status(200).send({msg:"Funciono Perfecto !"})
26 }
```

- **const salt = bcrypt.genSaltSync(10):** Esta línea usa la función `bcrypt.genSaltSync()` para generar una sal de 10 bytes. La función `bcrypt.genSaltSync()` toma un número como argumento, que indica la longitud de la sal. En este caso, estamos usando un número de 10, que es una longitud de sal segura.
- **const hashPassword = bcrypt.hashSync(password, salt):** Esta línea usa la función `bcrypt.hashSync()` para encriptar la contraseña con la sal. La función `bcrypt.hashSync()` toma dos argumentos: la contraseña y la sal. En este caso, estamos usando la contraseña `password` y la sal que generamos en la línea anterior.

Se le llama sal a la cadena aleatoria que se usa para encriptar la contraseña porque, al igual que la sal en la cocina, agrega un toque de sabor y complejidad al hash de la contraseña. La sal hace que el hash sea más difícil de descifrar, incluso si un atacante conoce la contraseña original.

La sal se usa en el proceso de encriptación para mezclar la contraseña con una cadena aleatoria. Esto hace que sea más difícil para un atacante adivinar la contraseña original, incluso si conoce el hash.

La sal también se usa para evitar ataques de diccionario. Los ataques de diccionario son un tipo de ataque de fuerza bruta que utilizan una lista de contraseñas comunes. Si un atacante conoce la sal, puede usarla para crear una lista de contraseñas encriptadas que coincidan con la contraseña original. Sin embargo, si la sal es aleatoria, es mucho más difícil para un atacante crear una lista de contraseñas encriptadas que coincidan.

Ahora que conocemos como funciona activamos nuestro servidor de nuevo y hacemos el envío de la información a través de insomnia para ver lo que nos muestra nuestro log:

```
[nodemon] starting `node index.js`
#####
#####
### API REST MERN #####
#####
http://localhost:3000/api/v1/
123456
$2a$10$MKQ1J6z0Ag/98YYCwgUy20UkDs3EGYEwLxL0z945NtiLUTs0AYwi6
```



@hdtoledo

Acá observamos como nuestra contraseña pasa a ser segura a través de **bcrypt**. Ahora colocamos nuestra contraseña encriptada dentro de nuestro objeto de datos de la siguiente manera y hacemos el log con **user**:

```
19     const salt = bcrypt.genSaltSync(10)
20     const hashPassword = bcrypt.hashSync(password, salt)
21
22     user.password = hashPassword
23
24     console.log(user)
25
26     res.status(200).send({msg:"Funciono Perfecto !"})
```

Si verificamos de nuevo con insomnia enviando los datos en nuestra consola tendremos lo siguiente:

```
http://localhost:3000/api/v1/
{
  firstname: 'Jeronimo',
  lastname: 'Hernandez Cortes',
  email: 'elinsano2023@gmail.com',
  role: 'User',
  active: false,
  _id: new ObjectId("65163372bafac5255a0ca660"),
  password: '$2a$10$MH.qn.T24z10guuYa3ohyuhW3ph1lkwsbec1jfPzsnsrgh/D2nbuoq'}
```

Ya que nuestra contraseña esta correctamente encriptada, ahora lo que nos queda es realizar el registro verdadero y para ello lo que vamos a hacer es lo siguiente:

```
19     const salt = bcrypt.genSaltSync(10)
20     const hashPassword = bcrypt.hashSync(password, salt)
21
22     user.password = hashPassword
23
24     user.save((error, userStorage) => {
25       if (error){
26         res.status(400).send({msg: "Error al crear el usuario"})
27       } else {
28         res.status(200).send(userStorage)
29       }
30     })
31
32
33   module.exports = {
```

- **user.save((error, userStorage) => {})**: Esta línea guarda el usuario en la base de datos y devuelve una promesa. La promesa se resuelve con el usuario guardado si la operación es exitosa, o con un error si la operación falla.
- **if (error){}**: Esta línea comprueba si el error es nulo. Si el error es nulo, el código envía el usuario guardado al cliente con un código de estado 200.
- **res.status(400).send({msg: "Error al crear el usuario"})**: Esta línea envía un mensaje de error al cliente con un código de estado 400.
- **else {}**: Esta línea se ejecuta si el error no es nulo.
- **res.status(200).send(userStorage)**: Esta línea envía el usuario guardado al cliente con un código de estado 200.

Ahora lo que nos queda es probar el registro realizando el envío de datos a través de insomnia, ejecutamos y en consola nos debe salir lo siguiente:



@hdtoledo

```

1: {
2:   "firstname": "Jimmy",
3:   "lastname": "Lombana",
4:   "email": "jimmy@gmail.com",
5:   "password": "123456",
6:   "_id": "65172aac6ebdfb77523a08d4",
7:   "role": "user",
8:   "active": false,
9:   "password": "$2a$10$VLt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG",
10:  "__v": 0
}

```

Todo debe salir correctamente y si verificamos la consola no debemos tener errores:

```

LAPTOPHDMI ~ D:\DATA\...\server yarn 18.16.0 1ms
{ hdtledo } yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1

```

Ahora vamos a mongodb y revisamos en atlas en nuestra DB que tendremos unos datos ya registrados:

HECTOR DAVID'S ORG - 2023-04-25 > PROJECT 0 > DATABASES

mern-web-personal

- [Overview](#)
- [Real Time](#)
- [Metrics](#)
- [Collections](#) (selected)
- [Search](#)
- [Profiler](#)
- [Performance Advisor](#)
- [Online Archive](#)
- [Cmd Line Tools](#)

DATABASES: 1 COLLECTIONS: 1

[+ Create Database](#)

test.users

STORAGE SIZE: 20KB LOGICAL DATA SIZE: 200B TOTAL DOCUMENTS: 1 INDEXES TOTAL SIZE: 40KB

[Find](#) [Indexes](#) [Schema Anti-Patterns](#) [Aggregation](#) [Search Indexes](#)

Filter Type a query: { field: 'value' }

QUERY RESULTS: 1-1 OF 1

```

_id: ObjectId('65172aac6ebdfb77523a08d4')
firstname: "Jimmy"
lastname: "Lombana"
email: "jimmy@gmail.com"
role: "user"
active: false
password: "$2a$10$VLt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG"
__v: 0

```



Si volvemos nuevamente a enviar el mismo registro nos debe salir el error de que ya está creado:

POST ▾ http://localhost:3000/api/v1/auth/register

Send ▾ 400 Bad Request | 174 ms | 35 B

Preview ▾ Headers Cookies Timeline

```
1 + {  
2   "firstname": "Jimmy",  
3   "lastname": "Lombana",  
4   "email": "jimmy@gmail.com",  
5   "password": "123456"  
6 }
```

```
1 + {  
2   "msg": "Error al crear el usuario"  
3 }
```

Y si cambiamos de usuario y registramos:

POST ▾ http://localhost:3000/api/v1/auth/register

Send ▾ 200 OK | 179 ms | 218 B

Preview ▾ Headers Cookies Timeline

```
1 + {  
2   "firstname": "Cristian",  
3   "lastname": "Lobaton",  
4   "email": "elpintor@gmail.com",  
5   "password": "123456"  
6 }
```

```
1 + {  
2   "firstname": "Cristian",  
3   "lastname": "Lobaton",  
4   "email": "elpintor@gmail.com",  
5   "role": "user",  
6   "active": false,  
7   "_id": "65172ce26ebdfb77523a08d8",  
8   "password": "$2a$10$FVoccCGlEAAvdcMm8PK9.PNxvsfstpkL3UnIkaQ8J6vpjOhjFIIM",  
9   "__v": 0  
10 }
```

Y revisamos en atlas:

Data services App Services Charts

Overview Real Time Metrics Collections Search Profiler Performance Advisor Online Archive Cmd Line Tools

DATABASES: 1 COLLECTIONS: 1

+ Create Database

Q Search Namespaces

test

users

test.users

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 406B TOTAL DOCUMENTS: 2 INDEXES TOTAL SIZE: 72KB

Find Indexes Schema Anti-Patterns Aggregation Search Indexes

Filter Type a query: { field: 'value' } Reset Apply More Options ▾

QUERY RESULTS: 1-2 OF 2

```
_id: ObjectId('65172aa6ebdfb77523a08d4')  
firstname: "Jimmy"  
lastname: "Lombana"  
email: "jimmy@gmail.com"  
role: "user"  
active: false  
password: "$2a$10$Vlt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMwNib./CEHD7ltgMppG"  
__v: 0
```

```
_id: ObjectId('65172ce26ebdfb77523a08d8')  
firstname: "Cristian"  
lastname: "Lobaton"  
email: "elpintor@gmail.com"  
role: "user"  
active: false  
password: "$2a$10$FVoccCGlEAAvdcMm8PK9.PNxvsfstpkL3UnIkaQ8J6vpjOhjFIIM"  
__v: 0
```

De esta manera ya tenemos nuestro registro de usuarios configurado.



ACCESS TOKEN Y REFRESH TOKEN

Los "Access Tokens" y "Refresh Tokens" son dos conceptos fundamentales en el ámbito de la autenticación y autorización en aplicaciones web y servicios. Se utilizan comúnmente en sistemas de autenticación y autorización basados en tokens, como OAuth 2.0 y JSON Web Tokens (JWT), para garantizar la seguridad y la gestión de sesiones en aplicaciones web y APIs.

Access Token (Token de Acceso):

1. **Propósito:** Un Access Token es un token de seguridad que se utiliza para autorizar y autenticar a un usuario o una aplicación cuando intenta acceder a recursos protegidos, como datos de usuario o servicios web.
2. **Duración:** Por lo general, los Access Tokens tienen una vida útil relativamente corta y están diseñados para ser válidos solo durante un período breve (pocos minutos u horas).
3. **Uso:** Un cliente (aplicación o usuario) incluye el Access Token en cada solicitud que hace a un servidor para acceder a recursos protegidos. El servidor verifica la validez y los permisos del token antes de permitir el acceso al recurso solicitado.
4. **Renovación:** Debido a su corta vida útil, los Access Tokens a menudo deben renovarse o renovarse antes de que expiren. Esto puede hacerse utilizando el Refresh Token.
5. **Ejemplo:** En una aplicación web que utiliza OAuth 2.0, un usuario inicia sesión y obtiene un Access Token que se incluye en cada solicitud API posterior para acceder a su información de perfil.

Refresh Token (Token de Actualización o Renovación):

6. **Propósito:** Un Refresh Token es un token de seguridad que se utiliza para obtener un nuevo Access Token cuando el Access Token original ha expirado o está a punto de expirar.
7. **Duración:** Los Refresh Tokens suelen tener una vida útil más larga que los Access Tokens, lo que les permite ser válidos durante días o incluso semanas.
8. **Uso:** Cuando un Access Token ha expirado o está cerca de su vencimiento, el cliente (aplicación o usuario) puede utilizar el Refresh Token para solicitar un nuevo Access Token sin necesidad de autenticarse nuevamente. Esto evita que el usuario tenga que volver a ingresar sus credenciales.
9. **Seguridad:** Debido a que los Refresh Tokens tienen una vida útil más larga, deben manejarse con cuidado y almacenarse de manera segura. Los Refresh Tokens son una forma importante de mitigar riesgos de seguridad al evitar que los Access Tokens tengan una vida útil demasiado larga.
10. **Ejemplo:** Un usuario inicia sesión en una aplicación móvil y obtiene tanto un Access Token como un Refresh Token. Cuando el Access Token expira, la aplicación utiliza el Refresh Token para obtener un nuevo Access Token sin requerir que el usuario vuelva a ingresar sus credenciales.



@hdtoledo

Bien para aplicar nuestro control vamos a dirigirnos a **yarn** y ubicamos **JsonWebToken**:

The screenshot shows the yarn package search interface. At the top, there's a navigation bar with links for 'Getting Started', 'Docs', 'Packages', and 'Blog'. On the right, there are social media sharing icons for English, GitHub, Twitter, Facebook, and LinkedIn. Below the navigation is a search bar with placeholder text 'Search packages (i.e. babel, webpack, react...)'. The main area is titled 'Package detail' and shows the 'jsonwebtoken' package. It includes the package name, version (8.5.1), license (MIT), and a brief description: 'JSON Web Token implementation (symmetric and asymmetric)'. There are also links for 'jwt' and 'examples'. To the right, there's a sidebar with links to other jsonwebtoken-related packages: 'yarn.pm/jsonwebtoken', 'auth0/node-jsonwebtoken', and 'npm/jsonwebtoken'. Below the sidebar, there are 'Install' and 'Edit' buttons.

Ejecutamos el comando **yarn add jsonwebtoken@8.5.1** :

```
LAPTOPHDMI1 D:\DATA\...\server yarn 18.16.0 1ms
{ hdtledo } yarn add jsonwebtoken@8.5.1
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

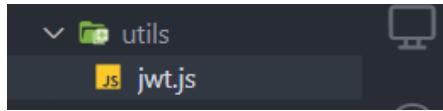
success Saved lockfile.
success Saved 12 new dependencies.
info Direct dependencies
└─ jsonwebtoken@8.5.1
info All dependencies
├─ buffer-equal-constant-time@1.0.1
├─ ecdsa-sig-formatter@1.0.11
├─ jsonwebtoken@8.5.1
├─ jwa@1.4.1
├─ jws@3.2.2
├─ lodash.includes@4.3.0
├─ lodash.isboolean@3.0.3
├─ lodash.isinteger@4.0.4
├─ lodash.isnumber@3.0.3
├─ lodash.isplainobject@4.0.6
└─ lodash.isstring@4.0.1
└─ lodash.once@4.1.1
Done in 1.47s.
```

Revisamos en nuestras dependencias:

```
12 "dependencies": [
13   "bcryptjs": "^2.4.3",
14   "body-parser": "1.20.0",
15   "cors": "2.8.5",
16   "express": "4.18.1",
17   "jsonwebtoken": "8.5.1",
18   "mongoose": "6.6.1",
19   "nodemon": "2.0.20"
20 ]
21 }
```



Ahora vamos a crear el archivo **jwt.js** dentro de **utils** y allí vamos a colocar todo lo que tenga que ver con la creación y demás de nuestro **json web token**:



Y ahora vamos a configurar de la siguiente manera:

```
utils > js jwt.js > ...
1  const jwt = require("jsonwebtoken")
2
3
```

Vamos a nuestro archivo **constants.js** y agregamos una nueva variable llamada **JWT_SECRET_KEY**, el contenido de esta va a ser aleatorio así que pueden perfectamente escribir lo que deseen, y la exportamos junto con las demás:

```
6  const IP_SERVER = "localhost"
7  const JWT_SECRET_KEY = "lkajsdf8967sd09f09453fgh45j3k786as0doif98S7DF908"
8
9  module.exports = {
10    DB_USER,
11    DB_PASSWORD,
12    DB_HOST,
13    API_VERSION,
14    IP_SERVER,
15    JWT_SECRET_KEY
16 }
```

Ahora volvemos a **jwt.js** y hacemos la importación de nuestra nueva variable:

```
utils > js jwt.js > ...
1  const jwt = require("jsonwebtoken")
2  const { JWT_SECRET_KEY } = require("../constants")
3
```

Ahora creamos la función:

```
utils > js jwt.js > ↗ createAccessToken
1  const jwt = require("jsonwebtoken")
2  const { JWT_SECRET_KEY } = require("../constants")
3
4  function createAccessToken(user){
5    const expirationToken = new Date()
6    expirationToken.setHours(expirationToken.getHours() + 3)
7
8    const payload = {
9      token_type: "access",
10     user_id: user._id,
11     iat: Date.now(),
12     exp: expirationToken.getTime()
13   }
14
15   return jwt.sign(payload, JWT_SECRET_KEY)
16 }
```



@hdtoledo

1. Se importa la biblioteca **jsonwebtoken**, que es una biblioteca comúnmente utilizada para crear y verificar tokens JWT en aplicaciones Node.js.
2. Se importa la constante **JWT_SECRET_KEY** desde un archivo llamado "**../constants**". Esta constante debería contener la clave secreta que se utiliza para firmar y verificar los tokens JWT. La clave secreta es esencial para garantizar la seguridad de los tokens.
3. La función **createAccessToken** acepta un argumento **user**, que se supone que es el objeto de usuario para el cual se está creando el Access Token.
4. Se crea una variable llamada **expirationToken**, que representa la fecha y hora en la que el token expirará. En este caso, se establece la expiración en 3 horas después de la creación del token. Esto significa que el token será válido durante 3 horas a partir de su creación.
5. Luego, se construye un objeto **payload** que contiene la información que se incluirá en el token JWT. El objeto **payload** suele contener información relacionada con el usuario y detalles sobre el token. En este caso, el **payload** incluye:
 - **token_type**: Un campo que indica que este es un token de tipo "access".
 - **user_id**: El identificador único del usuario al que pertenece el token.
 - **iat** (tiempo de emisión): La marca de tiempo que indica cuándo se emitió el token (en milisegundos desde la época UNIX).
 - **exp** (tiempo de expiración): La marca de tiempo en la que el token expirará (en milisegundos desde la época UNIX).
6. Finalmente, se utiliza **jwt.sign()** para firmar el **payload** utilizando la clave secreta **JWT_SECRET_KEY** y se devuelve el token JWT resultante como resultado de la función.

Ahora creamos nuestro **RefreshToken** de la siguiente manera:

```

18 function createRefreshToken(user) {
19   const expirationToken = new Date()
20   expirationToken.getMonth(expirationToken.getMonth() + 1)
21
22   const payload = {
23     token_type: "refresh",
24     user_id: user._id,
25     iat: Date.now(),
26     exp: expirationToken.getTime()
27   }
28
29   return jwt.sign(payload, JWT_SECRET_KEY)
30 }
```

1. La función **createRefreshToken** toma un argumento **user**, que se supone que es el objeto de usuario para el cual se está creando el Refresh Token.
2. Se crea una variable llamada **expirationToken**, que representa la fecha y hora en la que el token expirará. En este caso, se crea una nueva instancia de la fecha actual y luego se llama a



@hdtoledo

`getMonth` para obtener el mes actual y se le suma 1. Esto se hace para establecer la expiración del token en 1 mes a partir de la creación del token.

3. Luego, se construye un objeto **payload** que contiene la información que se incluirá en el token JWT. Al igual que en el caso del Access Token, el **payload** suele contener información relacionada con el usuario y detalles sobre el token. En este caso, el **payload** incluye:
 - **token_type**: Un campo que indica que este es un token de tipo "refresh".
 - **user_id**: El identificador único del usuario al que pertenece el token.
 - **iat** (tiempo de emisión): La marca de tiempo que indica cuándo se emitió el token (en milisegundos desde la época UNIX).
 - **exp** (tiempo de expiración): La marca de tiempo en la que el token expirará, que se calcula en base al mes actual más uno (en milisegundos desde la época UNIX).
4. Finalmente, se utiliza `jwt.sign()` para firmar el **payload** utilizando la clave secreta `JWT_SECRET_KEY`, y se devuelve el token JWT resultante como resultado de la función.

Ahora creamos nuestra función para decodificar nuestro token:

```
31
32   function decoded(token) {
33     return jwt.decode(token, JWT_SECRET_KEY, true)
34   }
35
```

1. La función **decoded** toma un argumento llamado **token**, que se supone que es el token JWT que se desea decodificar.
2. Dentro de la función, se utiliza la función `jwt.decode()` de la biblioteca `jsonwebtoken` para decodificar el token. Los argumentos de esta función son:
 - **token**: El token JWT que se desea decodificar.
 - **JWT_SECRET_KEY**: La clave secreta que se utiliza para verificar la firma del token. Esta variable debe haber sido definida en algún lugar de tu código y debe contener la clave secreta necesaria para validar la firma del token.
 - **true**: Este tercer argumento indica que se debe verificar la firma del token. Cuando se establece en **true**, la biblioteca verificará la firma del token para asegurarse de que sea válido.
3. La función `jwt.decode()` decodifica el token y devuelve un objeto JavaScript que representa el contenido del token. Este objeto generalmente contiene los datos incluidos en el token, como el **payload** que contiene información sobre el usuario o la entidad a la que pertenece el token.
4. Finalmente, la función **decoded** devuelve el objeto decodificado como resultado.



Y por último creamos la exportación de nuestras funciones:

```
35
36 module.exports = [
37   createAccessToken,
38   createRefreshToken,
39   decoded
40 ]
```

LOGIN DE USUARIO:

Para crear nuestro login lo primero que vamos a realizar es ir a nuestro **auth.controller.js** y vamos a importar **jwt**:

```
controllers > js auth.controller.js > 📁 jwt
1 const bcrypt = require("bcryptjs")
2 const User = require("../models/user.model")
3 const jwt = require("../utils/jwt")
```

Y ahora nos ubicamos abajo para crear nuestra función:

```
31
32 function login(req, res){
33   const { email, password } = req.body
34   if(!email) res.status(400).send({msg: "El email es obligatorio"})
35   if(!password) res.status(400).send({msg: "La contraseña es obligatoria"})
36
37   const emailLowerCase = email.toLowerCase()
38
39   User.findOne({ email: emailLowerCase }, (error, userStore) => {
40     if(error){
41       res.status(500).send({msg: "Error del servidor"})
42     } else {
43       console.log("Password: ", password)
44       console.log(userStore)
45     }
46   })
47 }
```

1. La función **login** toma dos argumentos: **req** y **res**, que representan la solicitud (request) y la respuesta (response) HTTP, respectivamente. Esta función probablemente se utiliza como controlador de una ruta de inicio de sesión en tu servidor web.
2. Se desestructuran los datos de la solicitud **req.body** para obtener el **email** y la **password** proporcionados por el usuario en el formulario de inicio de sesión.
3. Se realiza una validación básica para verificar que se hayan proporcionado tanto el **email** como la **password**. Si falta alguno de estos campos, se responde con un código de estado HTTP 400 (Bad Request) y se envía un objeto JSON con un mensaje de error que indica qué campo falta.
4. Se convierte el **email** a minúsculas utilizando **toLowerCase()**. Esto se hace para asegurarse de que la búsqueda en la base de datos sea insensible a mayúsculas y minúsculas, ya que los correos electrónicos suelen ser insensibles a mayúsculas y minúsculas.

- Se utiliza `User.findOne()` para buscar un documento de usuario en la base de datos que coincida con el `email` proporcionado. Esta función espera una consulta de búsqueda y una función de devolución de llamada (callback) que se ejecutará una vez que se complete la búsqueda.
- En la función de devolución de llamada de `User.findOne()`, se maneja la respuesta de la búsqueda. Si ocurre un error durante la búsqueda, se responde con un código de estado HTTP 500 (Internal Server Error) y se envía un mensaje de error indicando un problema en el servidor.

Por último, si la búsqueda se realiza con éxito y se encuentra un usuario con el correo electrónico proporcionado, se muestra información de depuración en la consola. Esto incluye la contraseña proporcionada en el formulario de inicio de sesión (`password`) y los datos del usuario encontrado (`userStore`). Esto lo hacemos para validar que la información se nos esté mostrando, pero luego lo cambiaremos porque no debemos dejar esto así por seguridad.

Y como paso final exportamos login:

```

49
50   module.exports = {
51     register,
52     login
53   }
54

```

Ahora nos dirigimos a `auth.router.js` y creamos nuestro **endpoint** de login:

```

router > auth.router.js > ...
1  const express = require("express")
2  const AuthController = require("../controllers/auth.controller")
3
4  const api = express.Router()
5
6  api.post("/auth/register", AuthController.register)
7  api.post("/auth/login", AuthController.login)
8
9
10 module.exports = api

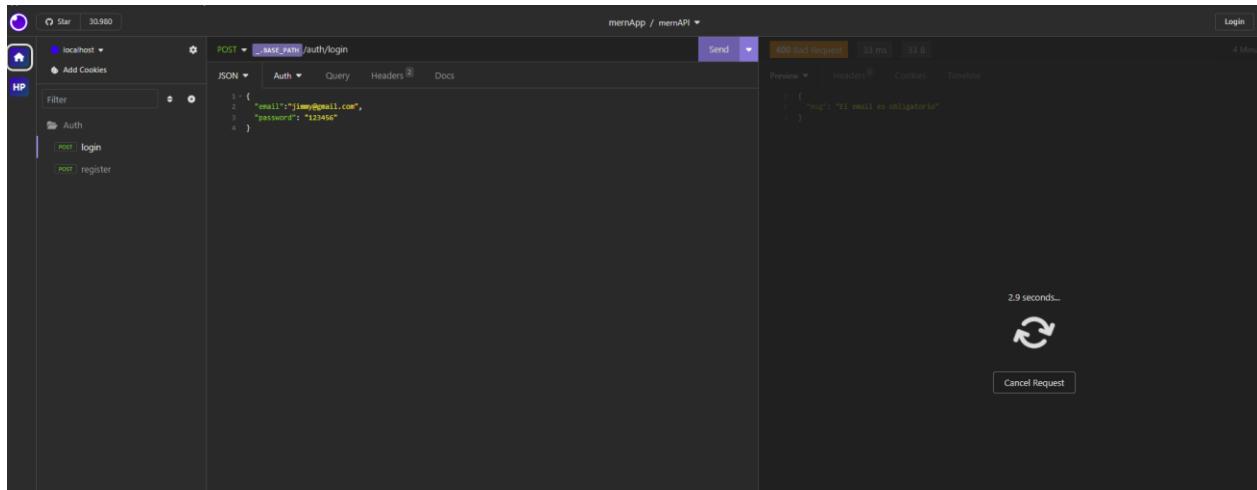
```

Ahora en nuestro **insomnia** vamos a crear un nuevo **endpoint** para el **login**:

The screenshot shows the Insomnia REST Client interface. On the left, there's a sidebar with a tree view showing an 'Auth' folder containing 'POST login' and 'POST register'. The main area has a 'POST' method selected, pointing to the URL '/auth/login'. The 'Headers' tab is active, showing 'Content-Type: application/json'. The 'Body' tab contains a JSON object: { "email": "" }. The 'Send' button is highlighted. To the right, the status bar shows 'mernApp / mernAPI' and '400 Bad Request'. Below it, the response pane shows the JSON object: { "msg": "El email es obligatorio" }.

Si hacemos una prueba sin enviar aun nada nos debe devolver el mail es obligatorio, ahora vamos a colocar la información de logeo:





Al enviar la información notamos que no recibimos nada de validación debido a que nuestra contraseña esta encriptada, si revisamos y comparamos en consola:

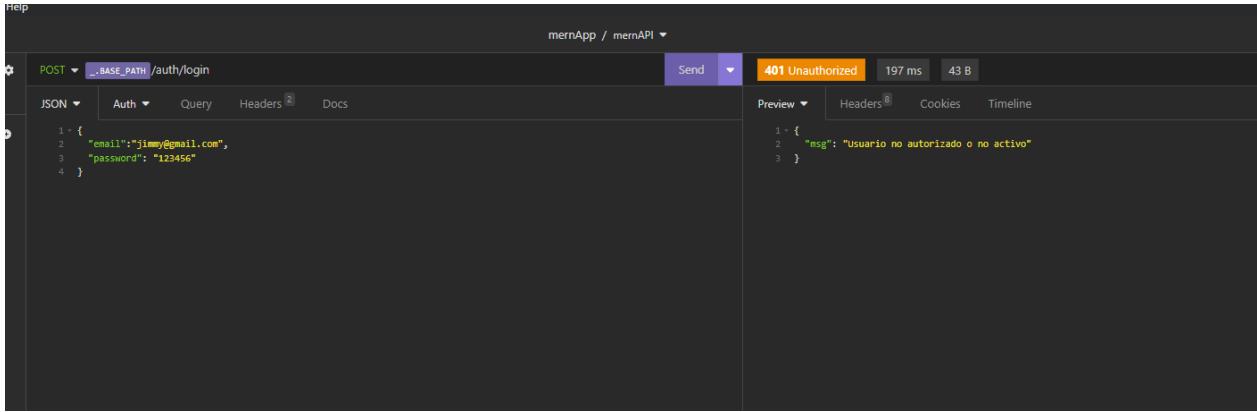
```
at next 11:11:11.621ms -> [object Object]
at Route.dispatch (D:\DATA\Documents\mern-app\server\node_modules\express\lib\router\index.js:42:14)
Password: 123456
{
  _id: new ObjectId("65172aac6ebdfb77523a08d4"),
  firstname: 'Jimmy',
  lastname: 'Lombana',
  email: 'jimmy@gmail.com',
  role: 'user',
  active: false,
  password: '$2a$10$VLt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG',
  __v: 0
}
```

Tenemos nuestro password sin encriptar y en la db encriptada a lo cual no se validarán para ello debemos hacer el proceso de desencriptar, entonces vamos a realizar lo siguiente en **aut.controller.js**:

```
controllers > auth.controller.js > ...
32   function login(req, res){
33     const { email, password } = req.body
34     if(!email) res.status(400).send({msg: "El email es obligatorio"})
35     if(!password) res.status(400).send({msg: "La contraseña es obligatoria"})
36
37     const emailLowerCase = email.toLowerCase()
38
39     User.findOne({ email: emailLowerCase }, (error, userStore) => {
40       if(error){
41         res.status(500).send({msg: "Error del servidor"})
42       } else {
43         bcrypt.compare(password, userStore.password, (bcryptError, check) => {
44           if (bcryptError) {
45             res.status(500).send({msg: "Error del servidor"})
46           } else if (!check) {
47             res.status(400).send({msg: "Usuario o Contraseña incorrecta"})
48           } else if (!userStore.active) {
49             res.status(401).send({msg: "Usuario no autorizado o no activo"})
50           } else {
51             res.status(200).send({msg: "Login Ok"})
52           }
53         })
54       }
55     })
56 }
```

Acá estamos haciendo las diferentes validaciones, la primera es la comparación de las contraseñas, y dentro de esta hacemos varias opciones de las cuales hacemos errores del servidor, del usuario o contraseña erróneas, si el usuario no está activo y por último si todo va bien enviamos un mensaje de ok.

Ahora vamos a nuestro insomnia a realizar la petición:



The screenshot shows the Insomnia REST Client interface. A POST request is made to `__BASE_PATH/auth/login`. The JSON body contains `{ "email": "jimmy@gmail.com", "password": "123456" }`. The response status is **401 Unauthorized**, with a response time of 197 ms and a body size of 43 B. The response message is `{ "msg": "Usuario no autorizado o no activo" }`.

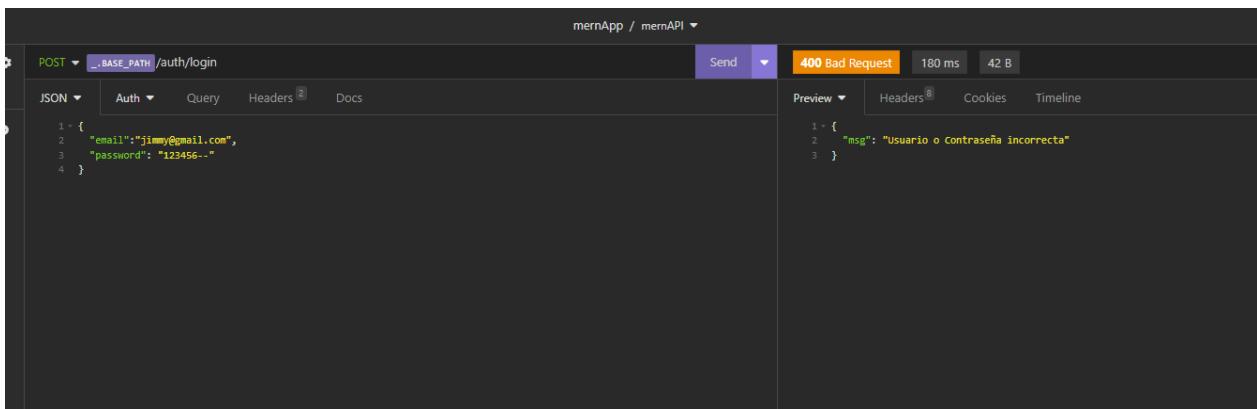
Y como resultado obtenemos usuario no autorizado o no activo, si revisamos nuestra db vamos a observar que el usuario no está activado:



QUERY RESULTS: 1-2 OF 2

```
1 _id: ObjectId('65172aac6ebdfb77523a08d4')
2 firstname: "Jimmy"
3 lastname: "Lombana"
4 email: "jimmy@gmail.com"
5 role: "user"
6 active: false
7 password: "$2a$10$VLt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG"
8 __v: 0
```

Ahora colocamos una contraseña incorrecta y enviamos:



The screenshot shows the Insomnia REST Client interface. A POST request is made to `__BASE_PATH/auth/login`. The JSON body contains `{ "email": "jimmy@gmail.com", "password": "123456--" }`. The response status is **400 Bad Request**, with a response time of 180 ms and a body size of 42 B. The response message is `{ "msg": "Usuario o Contraseña incorrecta" }`.

Ahora vamos a modificar nuestro usuario para que este activo, nos dirigimos a nuestra db y actualizamos:



QUERY RESULTS: 1-2 OF 2

```
_id: ObjectId('65172aac6ebdfb77523a08d4')
firstname: "Jimmy"
lastname: "Lombana"
email: "jimmy@gmail.com"
role: "user"
active: true
password: "$2a$10$VLt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG"
---v: 0
```

Nuevamente hacemos el envío de la información:

The screenshot shows a Postman interface with the following details:

- Method: POST
- URL: `http://localhost:3001/auth/login`
- Body (JSON):

```
1: {
2:   "email": "jimmy@gmail.com",
3:   "password": "123456"
4: }
```
- Status: 200 OK
- Time: 178 ms
- Size: 18 B
- Preview (Response):

```
1: {
2:   "msg": "Login Ok"
3: }
```

Y ahí obtenemos nuestro mensaje de que el login está ok.

Ahora vamos a modificar la ultima parte para que se nos generen los JWT tanto de Access como de Refresh:

```
48      } else if (!userStore.active) {
49        res.status(401).send({msg: "Usuario no autorizado o no activo"})
50      } else {
51        res.status(200).send({
52          access: jwt.createAccessToken(userStore),
53          refresh: jwt.createRefreshToken(userStore)
54        })
55      }
56    }
57  }
58}
59
60}
```

De esta manera si volvemos a enviar los datos de login, vamos a observar los tokens:



```

POST /auth/login
{
  "email": "jimmy@gmail.com",
  "password": "123456"
}

```

200 OK
199 ms 457 B
Just Now

```

{
  "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJ0b2tlbl90eXB1Ijo1YWNjZXNzIiwidXNlc19pZCI6IjY1MTcyYWFnMViZGZiNzc1MjNhMDhkNCIsIm1hdC16MTY5NjIwNjgxODYw0SwiZXhwIjoxNjk2MjE3NjE4NjA5fQ.EgpW99EW_EA1QmN15Egd7aYa8TwaEOWb2gwoPhYPBPM",
  "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJ0b2tlbl90eXB1Ijo1YWNjZXNzIiwidXNlc19pZCI6IjY1MTcyYWFnMViZGZiNzc1MjNhMDhkNCIsIm1hdC16MTY5NjIwNjgxODYw0SwiZXhwIjoxNjk2MjE3NjE4NjA5fQ.EgpW99EW_EA1QmN15Egd7aYa8TwaEOWb2gwoPhYPBPM"
}

```

Y cada vez que lo generemos se van a cambiar:

```

POST /auth/login
{
  "email": "jimmy@gmail.com",
  "password": "123456"
}

```

200 OK
163 ms 457 B
Just Now

```

{
  "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJ0b2tlbl90eXB1Ijo1YWNjZXNzIiwidXNlc19pZCI6IjY1MTcyYWFnMViZGZiNzc1MjNhMDhkNCIsIm1hdC16MTY5NjIwNjgxODYw0SwiZXhwIjoxNjk2MjE3NjE4NjA5fQ.EgpW99EW_EA1QmN15Egd7aYa8TwaEOWb2gwoPhYPBPM",
  "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJ0b2tlbl90eXB1Ijo1YWNjZXNzIiwidXNlc19pZCI6IjY1MTcyYWFnMViZGZiNzc1MjNhMDhkNCIsIm1hdC16MTY5NjIwNjgxODYw0SwiZXhwIjoxNjk2MjE3NjE4NjA5fQ.EgpW99EW_EA1QmN15Egd7aYa8TwaEOWb2gwoPhYPBPM"
}

```

Para validar el token es muy sencillo lo copiamos y nos dirigimos a la web jwt.io en donde lo validara:

Encoded
PASTE A TOKEN HERE

Decoded
EDIT THE PAYLOAD AND SECRET

Algorithm: HS256

HEADER: ALGORITHM & TOKEN TYPE

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

PAYOUT: DATA

```
{
  "token_type": "access",
  "user_id": "65172a6cebd677523a08d4",
  "iat": 1696206818609,
  "exp": 1696217618609
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

De esta manera dejamos nuestro login listo y funcional.



@hdtledo

REFRESCANDO EL ACCESS TOKEN

En esta parte vamos a refrescar el **Access token** una vez haya caducado, para ello lo enviamos mediante el **refresh token**, entonces nos dirigimos a **auth.controller.js** y creamos la función de **refreshAccessToken**:

```
controllers > js auth.controller.js > ...
60
61  function refreshAccessToken(req, res){
62    const { token } = req.body
63    const { user_id } = jwt.decoded(token)
64
65    User.findOne({ _id: user_id }, (error, userStorage) => {
66      if(error){
67        res.status(500).send({msg: "Error del servidor"})
68      } else {
69        res.status(200).send({
70          accessToken: jwt.createAccessToken(userStorage)
71        })
72      }
73    })
74  }
75 }
```

1. La función **refreshAccessToken** toma dos argumentos: **req** y **res**, que representan la solicitud (request) y la respuesta (response) HTTP, respectivamente. Esta función probablemente se utiliza como controlador de una ruta de renovación de tokens en tu servidor web.
2. Se desestructuran los datos de la solicitud **req.body** para obtener el **token**, que es el token JWT que se utilizará para solicitar la renovación del Access Token.
3. Luego, se utiliza **jwt.decoded(token)** para decodificar el token JWT que se proporcionó en la solicitud. Esto debería devolver un objeto que contiene información del usuario y otros datos incluidos en el token.
4. Se busca un usuario en la base de datos utilizando **User.findOne()**, donde se especifica que se desea encontrar un usuario cuyo **_id** coincida con el **user_id** extraído del token decodificado.
5. En la función de devolución de llamada de **User.findOne()**, se maneja la respuesta de la búsqueda. Si ocurre un error durante la búsqueda, se responde con un código de estado HTTP 500 (Internal Server Error) y se envía un mensaje de error indicando un problema en el servidor.
6. Si la búsqueda se realiza con éxito y se encuentra el usuario correspondiente, se crea un nuevo Access Token utilizando **jwt.createAccessToken(userStorage)**, donde **userStorage** es el objeto del usuario recuperado de la base de datos.
7. Finalmente, se responde con un código de estado HTTP 200 (OK) y se envía un objeto JSON que contiene el nuevo Access Token generado en la propiedad **accessToken**. Este nuevo Access Token se puede utilizar para autenticar y autorizar las solicitudes del usuario sin necesidad de volver a iniciar sesión.



@hdtoledo

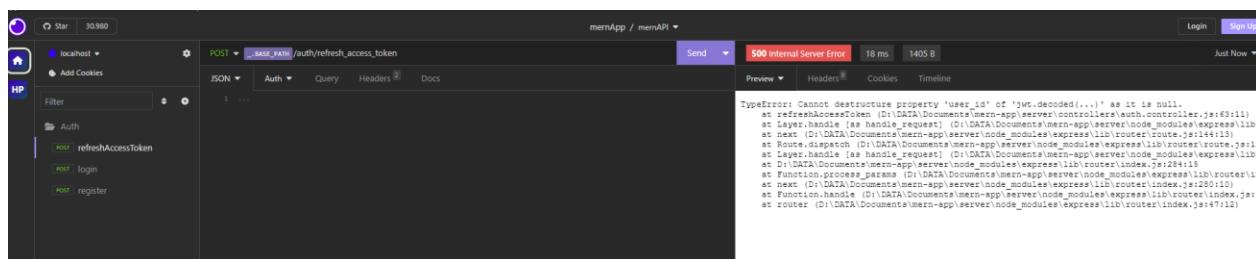
Por último, agregamos la exportación de nuestra función:

```
76
77 module.exports = {
78   register,
79   login,
80   refreshAccessToken
81 }
82
```

Ahora agregamos nuestra ruta en **auth.router.js**:

```
5
6   api.post("/auth/register", AuthController.register)
7   api.post("/auth/login", AuthController.login)
8   api.post("/auth/refresh_access_token", AuthController.refreshAccessToken)
9
10
11 module.exports = api
```

Y ahora vamos a insomnia y creamos nuestra petición:



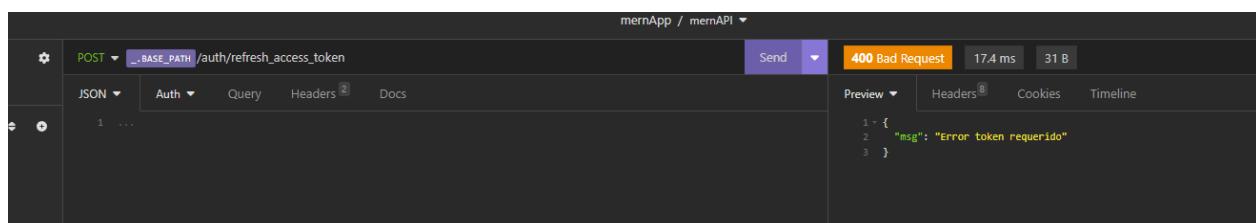
The screenshot shows the Insomnia REST Client interface. A POST request is being made to `/auth/refresh_access_token`. The response is a **500 Internal Server Error** with a timestamp of 18 ms and a body size of 1405 B. The error message in the preview tab is as follows:

```
TypeError: Cannot destructure property 'User_id' of 'jwt.decoded(..)' as it is null.
at refreshAccessToken (D:\DATA\Documents\mern-app\server\controllers\auth.controller.js:63:11)
at Layer.handle [as handle_request] (D:\DATA\Documents\mern-app\node_modules\express\lib\router\layer.js:95:12)
at Route.dispatch (D:\DATA\Documents\mern-app\node_modules\express\lib\router\route.js:114:13)
at Route.handle (D:\DATA\Documents\mern-app\node_modules\express\lib\router\route.js:11:12)
at Layer.handle [as handle_request] (D:\DATA\Documents\mern-app\server\index.js:28:10)
at Function.process_params (D:\DATA\Documents\mern-app\server\node_modules\express\lib\router\index.js:36:12)
at next (D:\DATA\Documents\mern-app\server\node_modules\express\lib\router\index.js:128:10)
at Function.handle (D:\DATA\Documents\mern-app\server\node_modules\express\lib\router\index.js:47:12)
at routes (D:\DATA\Documents\mern-app\server\node_modules\express\lib\router\index.js:71:12)
```

Si le damos a enviar solo por probar nos mostrara un error y es normal ya que no hemos definido un mensaje para el error así que vamos a crear uno en **auth.controller.js**:

```
61   function refreshAccessToken(req, res){
62     const { token } = req.body
63
64     if(!token) res.status(400).send({msg: "Error token requerido"})
65
66     const { user_id } = jwt.decoded(token)
67
68     User.findOne({ _id: user_id }, (error, userStorage) => {
```

Si volvemos a probar ya tendremos el mensaje:



The screenshot shows the Insomnia REST Client interface again. A POST request is being made to `/auth/refresh_access_token`. The response is a **400 Bad Request** with a timestamp of 17.4 ms and a body size of 31 B. The message in the preview tab is:

```
1: {
2:   "msg": "Error token requerido"
3: }
```

Y ahora vamos a probar a pasarle nuestro token para ello nos dirigimos en insomnia a nuestro login y copiamos el refresh token:



Y lo vamos a colocar en la solicitud de `refreshAccessToken`:

De esta manera nos fijamos que los tokens son diferentes y que automáticamente hizo el refresh token y está funcionando.

ESTRUCTURA API USER

Acá vamos a obtener, crear, eliminar y todo lo referente a nuestros usuarios, esto lo realizamos a través de los **endpoints** protegidos, ya que solo los usuarios registrados podrán realizarlo, para ello un middleware lo realizará, así que nos vamos a crear un nuevo controlador dentro de la carpeta **controllers** y su nombre será **user.controller.js** y allí colocaremos lo siguiente:

```

controllers > JS user.controller.js ...
1  async function getMe(req, res) {
2      res.status(200).send({msg: "Ok"})
3  }
4
5  module.exports = {
6      ...getMe
7  }

```

Ahora creamos la ruta, vamos a crear dentro de **router** nuestro archivo **user.router.js** y lo dejamos así:



```

router > js user.router.js > ...
1 const express = require("express")
2 const UserController = require("../controllers/user.controller")
3
4 const api = express.Router()
5
6 api.get("/user/me", UserController.getMe)
7 |
8 module.exports = api

```

Ahora nos vamos para **app.js** y dentro vamos a importar nuestra ruta:

```

8
9 // Importar rutas
10 const authRoutes = require("./router/auth.router")
11 const userRoutes = require("./router/user.router")
12
// Configuración Rutas

```

Y ahora configuramos allí mismo la ruta:

```

22
23 // Configurar Rutas
24 app.use(`/api/${API_VERSION}`, authRoutes)
25 app.use(`/api/${API_VERSION}`, userRoutes)
26
// Configuración Rutas

```

Si todo esta correcto podemos utilizar nuestra nueva ruta, para ello vamos a insomnia y creamos una nueva carpeta llamada **user** y hacemos una nueva solicitud tipo **GET** y nos debe devolver nuestro mensaje de Ok:

The screenshot shows the Insomnia API client interface. On the left, there's a sidebar with a list of endpoints under the 'User' category, including 'GetMe' (GET). In the main area, a request is being made to 'GET /user/me'. The response details show a green '200 OK' status, a response time of '9.88 ms', and a body size of '12 B'. The response preview pane displays the JSON object: { "msg": "OK" }.

Vamos a dejar hasta acá la configuración base de nuestro user.



MIDDLEWARE DE AUTENTICACION

Un middleware de autenticación es un componente de software que se utiliza en el desarrollo de aplicaciones web y servicios para verificar la identidad de un usuario o entidad que realiza una solicitud antes de permitir el acceso a ciertos recursos o funcionalidades. Su función principal es garantizar que solo los usuarios autenticados y autorizados puedan acceder a ciertas partes de una aplicación o servicio.

Aquí hay una descripción más detallada de cómo funciona un middleware de autenticación:

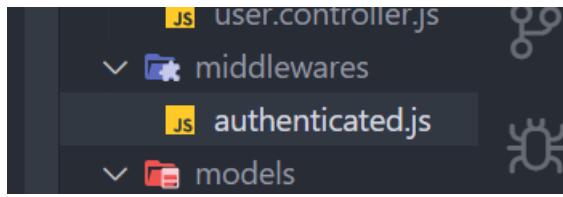
1. **Intercepción de Solicitudes:** El middleware de autenticación se coloca en el flujo de procesamiento de solicitudes HTTP entre el cliente y el servidor. Cuando un cliente realiza una solicitud a una ruta o recurso protegido, el middleware de autenticación interviene antes de que la solicitud alcance la función controladora correspondiente.
2. **Verificación de Identidad:** El middleware de autenticación verifica la identidad del cliente, comúnmente a través de credenciales como un nombre de usuario y una contraseña, tokens de acceso, certificados digitales u otros mecanismos de autenticación. Esta verificación puede implicar consultar una base de datos de usuarios, verificar la validez de un token, o realizar otro proceso de autenticación.
3. **Generación de Contexto de Usuario:** Una vez que se ha autenticado al cliente, el middleware de autenticación puede generar un "contexto de usuario". Este contexto de usuario suele contener información relevante sobre el cliente autenticado, como su identificador, roles y permisos.
4. **Autorización:** Además de la autenticación, algunos middlewares de autenticación también manejan la autorización. Esto implica verificar si el usuario autenticado tiene permisos para acceder al recurso o realizar la acción solicitada. Si el usuario no está autorizado, se le deniega el acceso.
5. **Redirección o Respuesta de Error:** Si la autenticación falla o el usuario no está autorizado, el middleware puede redirigir al usuario a una página de inicio de sesión, mostrar un mensaje de error o tomar otras acciones definidas por el desarrollador.

Los middlewares de autenticación son esenciales para garantizar la seguridad de una aplicación al proteger recursos y funcionalidades sensibles. Se utilizan en una variedad de contextos, como aplicaciones web, API REST, aplicaciones móviles y servicios en la nube. Al implementar un middleware de autenticación sólido, puedes asegurarte de que solo los usuarios legítimos tengan acceso a los datos y las funcionalidades que están autorizados a utilizar.

Para realizar nuestro CRUD debemos tener protegidos los **endpoints**, es decir que solo los usuarios registrados podrán acceder a estas funciones y para ello vamos a utilizar el middleware para que valide que el usuario está registrado y le permita ejecutar las funciones a través de los **endpoints** protegidos.

Nos dirigimos a nuestra carpeta **middlewares** y allí creamos el archivo **authenticated.js** y hacemos lo siguiente:





```
middlewares > JS authenticated.js > ...
1 const jwt = require("jsonwebtoken")
2
3 function assureAuth(req, res, next) {
4     console.log("Hola estoy en assure auth")
5     next()
6 }
7
8 module.exports = {
9     assureAuth
10 }
```

Aunque en este código no vamos a ejecutar validaciones importantes vamos a entender su funcionamiento y para ello vamos a dirigirnos a **user.router.js** y lo vamos a importar y ejecutar:

```
router > JS user.router.js > md_auth
1 const express = require("express")
2 const UserController = require("../controllers/user.controller")
3 const md_auth = require("../middlewares/authenticated")
4
```

Y ahora lo vamos a colocar en medio de la ejecución de nuestra ruta como un array, recuerden que podremos colocar no solo uno sino varios middlewares que se encarguen de validar nuestras rutas:

```
4
5 const api = express.Router()
6
7 api.get("/user/me", [md_auth.assureAuth], UserController.getMe)
8
9 module.exports = api
```

Y ahora nos dirigimos a nuestro insomnia y realizamos una solicitud nuevamente a **/user/me**

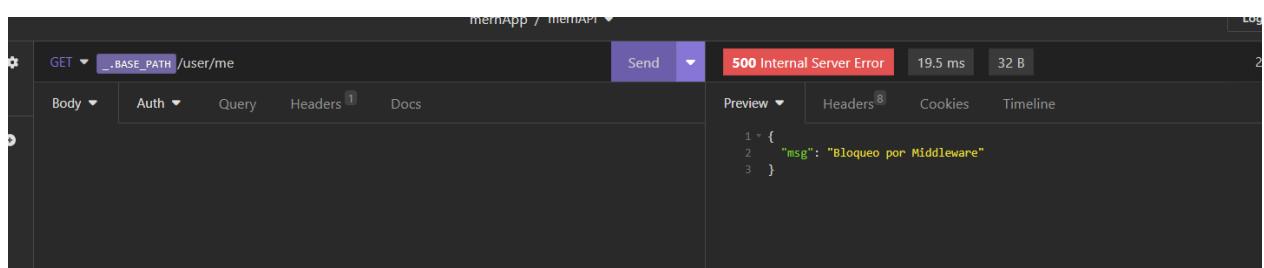
Hasta ahí todo va normal, pero si revisamos en consola obtendremos el mensaje:

```
[nodemon] starting - node index.js
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
Hola estoy en asure auth
```

Vamos a colocarle un status 500 para observar el mensaje en insomnia:

```
middlewares > js authenticated.js > asureAuth > msg
1 const jwt = require("jsonwebtoken")
2
3 function asureAuth(req, res, next) {
4   console.log("Hola estoy en asure auth")
5   res.status(500).send({msg: "Bloqueo por Middleware"})
6   next()
7 }
8
9 module.exports = {
10   asureAuth
11 }
```

Y en nuestro insomnia observamos lo siguiente:



The screenshot shows the insomnia API client interface. A GET request is made to `/_BASE_PATH/user/me`. The response status is **500 Internal Server Error**, with a response time of 19.5 ms and a body size of 32 B. The Headers tab shows the response headers. The Preview tab displays the JSON response body: `{ "msg": "Bloqueo por Middleware" }`.

De esta manera ya entendemos que va a realizar nuestro middleware, así que vamos a colocarle la lógica a nuestra función de la siguiente manera:

```
middlewares > js authenticated.js > ...
1 const jwt = require("jsonwebtoken")
2
3 function asureAuth(req, res, next) {
4   console.log(req.headers.authorization)
5   next()
6 }
7
8 module.exports = {
9   asureAuth
10 }
```

Vamos a utilizar `req` para poder hacer el envío y así acceder a la autorización, lo colocamos en un log para observar que nos trae, si damos a enviar la petición en insomnia y revisamos la consola nos saldrá lo siguiente:



@hdtoledo

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
undefined
```

Nos dice que no esta definido para lo cual debemos enviarle el token que creamos a través de insomnia:

The screenshot shows the Insomnia interface. On the left, there's a sidebar with a tree view of API endpoints: User (GetMe, refreshAccessToken), Auth (login, register). The main area shows a POST request to `/auth/login`. The JSON body is set to:

```
1: {
  2:   "email": "jimmy@gmail.com",
  3:   "password": "123456"
  4: }
```

The response tab shows a 200 OK status with a large JWT token displayed.

Lo colocamos en los **headers** y lo dejamos de la siguiente manera:

The screenshot shows the Insomnia interface. On the left, there's a sidebar with a tree view of API endpoints: User (GetMe, refreshAccessToken), Auth (login, register). The main area shows a GET request to `/user/me`. The Headers tab has an `Authorization` field set to `Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ...`. The response tab shows a 200 OK status with a message `"msg": "ok"`.

Bearer es un término utilizado en el contexto de autenticación y autorización en aplicaciones web y servicios. Se refiere a un tipo de token que se utiliza para identificar y autenticar a un usuario o una entidad que realiza una solicitud a un servidor.

En el contexto de la autenticación basada en tokens, como en el caso de JSON Web Tokens (JWT) o OAuth, "Bearer" se utiliza como parte de la cabecera de autorización de una solicitud HTTP. La cabecera de autorización tiene el siguiente formato:

Authorization: Bearer <token>

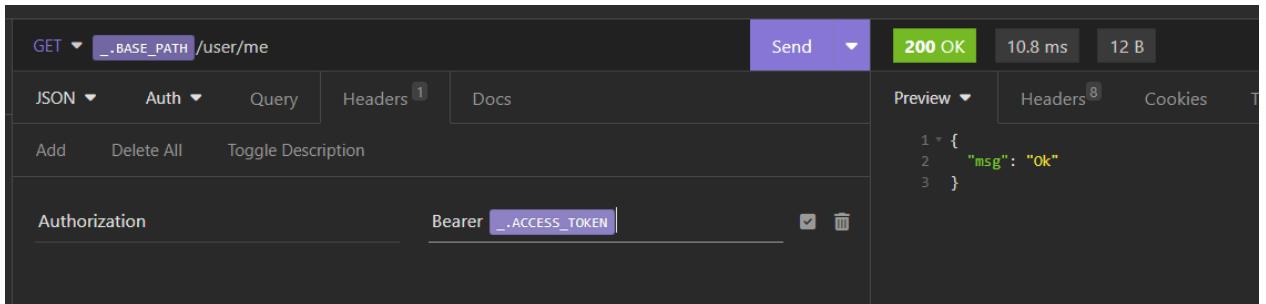
Al darle enviar vamos a obtener el token en consola:

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXAiOiAiYWNjZXNzIiwidXNlcI9pZCI6IjY1MTcyYWFiNmViZGZiNzc1MjNhMDhkNCIsImhdCI6MTY5Nj12MTU4MjY0NywiZXhwIjoaNjk2MjcyMzgyNjQ3fQ.33UU8sa18C2wKnJ9CnxOPSladOyIKsxUN0uzF5Z3xfg|
```

De esta manera ya lo vemos reflejado, pero vamos a automatizar un poco el proceso para no tener que estar copiando y pegando, así que vamos a la configuración de las variables de entorno y vamos a agregar una que se llame **Access_Token** y pegamos nuestro token allí:



Cerramos y volvemos nuevamente a configurar de la siguiente manera:



Y si volvemos a ejecutar:

```

#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXB1Ijo1YWNjZXNzIiwidXNlc19pZCI6IjY1MTcyYWFjNmViZGZiNzc1MjNhMDhkNCIsImhdCI6MTY5NjI2MTU4MjY0NywiZXhwIjoxNjk2MjcyMzgyNjQ3fQ.33UU8sa18C2wKnJ9Cnx0PSlodDyikSxUN0uzF5Z3xfg
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXB1Ijo1YWNjZXNzIiwidXNlc19pZCI6IjY1MTcyYWFjNmViZGZiNzc1MjNhMDhkNCIsImhdCI6MTY5NjI2MTU4MjY0NywiZXhwIjoxNjk2MjcyMzgyNjQ3fQ.33UU8sa18C2wKnJ9Cnx0PSlodDyikSxUN0uzF5Z3xfg
  
```

Ya lo estaremos viendo reflejado allí, ahora vamos a colocar lo siguiente en nuestra función para poder validar nuestra autenticación:

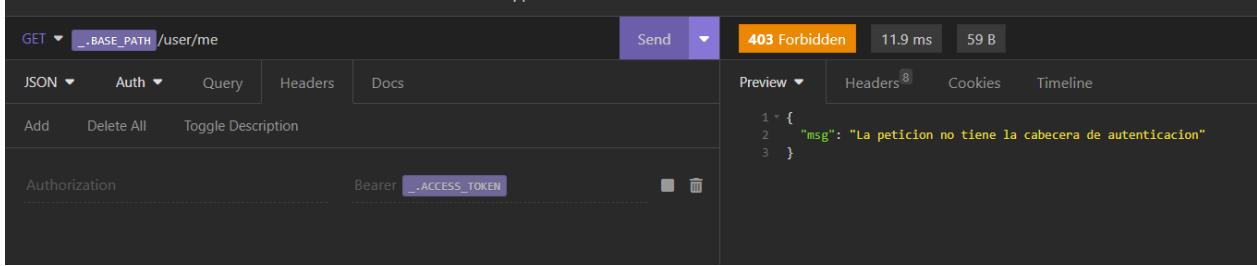
```

middlewares > js authenticated.js > ...
1  const jwt = require("jsonwebtoken")
2
3  function assureAuth(req, res, next) {
4    if(!req.headers.authorization){
5      res.status(403).send({msg: "La petición no tiene la cabecera de autenticación"})
6    }
7    next()
8  }
9
10 module.exports = {
11   assureAuth
12 }
  
```



@hdtoledo

Ahora hagamos el ejercicio de nuevo en insomnia para validar que está saliendo el error, quitamos el check y enviamos:



The screenshot shows the Insomnia REST Client interface. A GET request is made to `_BASE_PATH/user/me`. The response is a 403 Forbidden status with a duration of 11.9 ms and a size of 59 B. The response body is a JSON object with one key, `msg`, which contains the message "La peticion no tiene la cabecera de autenticacion".

Nos muestra el error que acabamos de configurar y va todo perfecto hasta acá, ahora lo que realizaremos será extraer nuestro token para poderlo procesar y para ello vamos a realizar lo siguiente:

```
middlewares > js authenticated.js > asureAuth
1 const jwt = require("jsonwebtoken")
2
3 function asureAuth(req, res, next) {
4   if(!req.headers.authorization){
5     res.status(403).send({msg: "La peticion no tiene la cabecera de autenticacion"})
6   }
7
8   const token = req.headers.authorization.replace("Bearer", "")
9   console.log(token)
10
11   next()
12 }
13
14 module.exports = {
15   asureAuth
16 }
```

Almacenamos en nuestra **const** el valor del token y remplazamos el **Bearer** que viene dentro por ningún **string**, ejecutamos el log para revisar que va limpio el token:

```
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXB1IjoiYWNjZXNzIiwidXNlcl9pZCI6IjY1MTcyYWFiNmViZGZiNzc1MjNhMDhkNCIsImhdCI6MTY5NjI2MTU4MjY0NywiZXhwIjoxNjk2MjcyMzgyNjQ3fQ.33UU8sa18C2wKnJ9CnxOPSlodOyIKsxUN0uzF5Z3xfg
```

Ahora que validamos que esta correcto nuestro token, modificamos de la siguiente manera colocando un try catch:



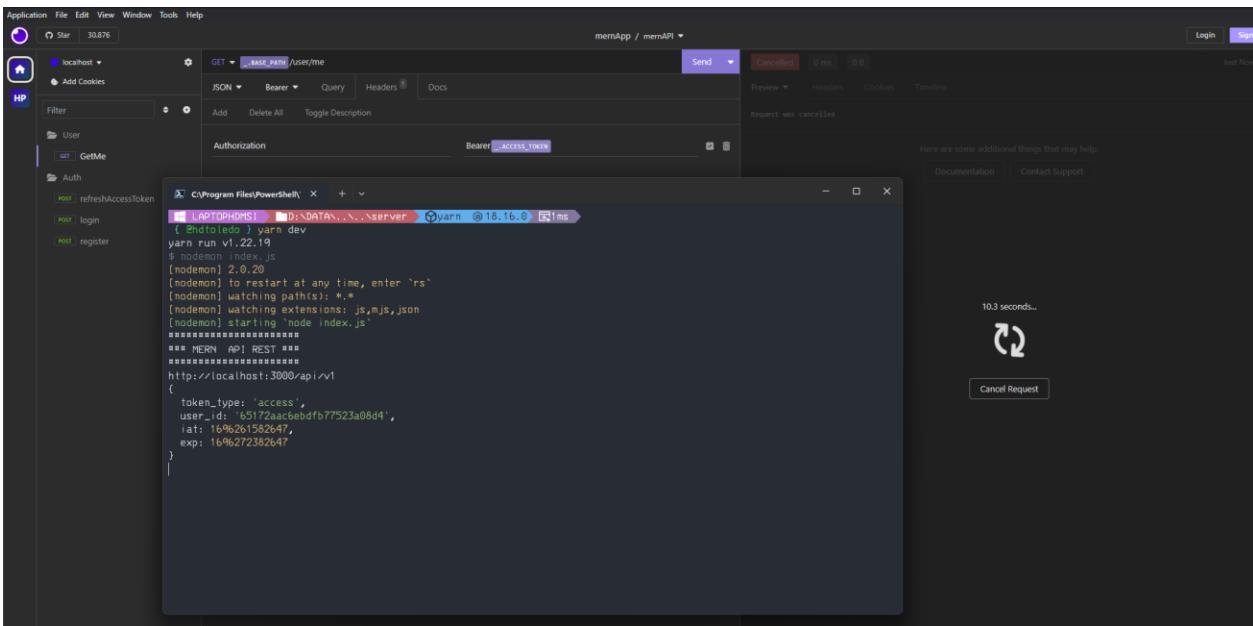
```

middlewares > js authenticated.js > asureAuth
1 const jwt = require("../utils/jwt")
2
3 function asureAuth(req, res, next) {
4   if(!req.headers.authorization){
5     return res.status(403).send({msg: "La peticion no tiene la cabecera de autenticacion"})
6   }
7
8   const token = req.headers.authorization.replace("Bearer", "")
9
10  try {
11    const payload = jwt.decoded(token)
12    console.log(payload);
13  } catch (error) {
14    return res.status(400).send({msg: "Token Invalido"})
15  }
16}
17
18 module.exports = {
19   asureAuth
20 }

```

De esta manera cambiamos en jwt nuestro archivo jwt, modificamos asureAuth y agregamos el return, y en el try catch cargamos nuestro payload con la información del token si ocurre un error nos mostrara el error del token invalido.

Vamos a probar de nuevo enviando la petición a través de insomnia:



Y observamos que ya tenemos los datos que necesitamos, ahora realizaremos la extracción de iat y la fecha de la siguiente manera:



```

7
8     const token = req.headers.authorization.replace("Bearer", "")
9
10    try {
11        const payload = jwt.decoded(token)
12
13        const { exp } = payload
14        const currentDate = new Date().getTime()
15
16        console.log(exp)
17        console.log(currentDate)
18
19    } catch (error) {
20        return res.status(400).send({msg: "Token Invalido"})
21    }
22}
23

```

Lo que hicimos fue hacer una extracción de la data que necesitamos para validar, mediante exp y currentDate y por ultimo los revisamos en consola, ejecutamos de nuevo insomnia y obtenemos lo siguiente:

```

[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
1696272382647
1696872653294
|

```

Ahora comprobaremos que la fecha de expiración no sea menor o igual a la fecha actual, si la fecha de exp es anterior a la fecha actual significa que ha caducado el token y si es superior que no ha caducado, para ello vamos a realizar lo siguiente:

```

10    try {
11        const payload = jwt.decoded(token)
12
13        const { exp } = payload
14        const currentDate = new Date().getTime()
15
16        console.log(exp)
17        console.log(currentDate)
18
19        if(exp <= currentDate){
20            return res.status(400).send({msg: "El token ha expirado"})
21        }
22
23        req.user = payload
24        next()
25
26    } catch (error) {
27        return res.status(400).send({msg: "Token Invalido"})
28    }
29}
30

```



@hdtoledo

Ahora comprobamos mediante insomnia que todo vaya bien, si ejecutamos nos sale lo siguiente:

The screenshot shows the Insomnia REST client interface. A GET request is made to `$_BASE_PATH/_user/me`. The Authorization header is set to `Bearer $_ACCESS_TOKEN`. The response status is **200 OK**, with a response time of 13 ms and 12 B. The response body is a JSON object with a single key `"msg": "ok"`.

Ya nos deja hacer la petición correctamente y si desmarcamos de nuevo y enviamos nos saldrá lo siguiente:

The screenshot shows the Insomnia REST client interface. A GET request is made to `$_BASE_PATH/_user/me`. The Authorization header is set to `Bearer $_ACCESS_TOKEN`. The response status is **400 Bad Request**, with a response time of 1.58 ms and 24 B. The response body is a JSON object with a single key `"msg": "Token Invalido"`.

De esta manera lo podemos emplear en cualquier endpoint.

Obtener los datos del Usuario Logueado

Ahora vamos a terminar la función del `getMe` en donde devolveremos los datos de usuario y el token, así que realizaremos los siguiente en **nuestro user.controller.js**

```
controllers > js user.controller.js > <unknown>
1  async function getMe(req, res) {
2    console.log(req.user)
3    res.status(200).send({msg: "OK"})
4  }
5
6  module.exports = {
7    getMe,
8  }
```

De esta manera comprobamos que este devolviendo el objeto con los datos del usuario:

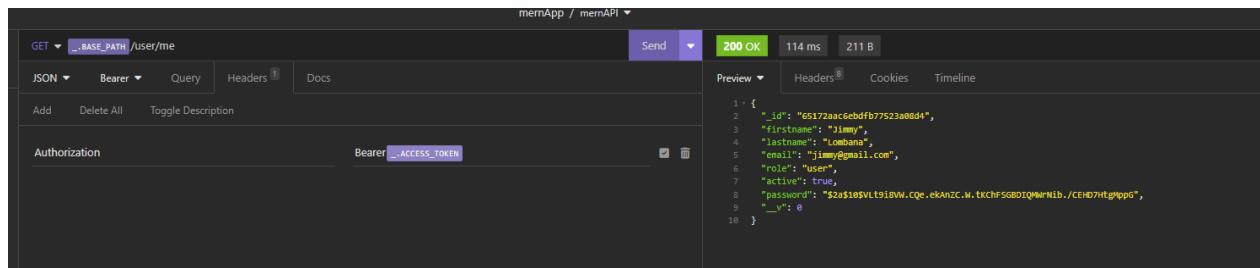
```
Lnodemini$ starting node index.js
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
{
  token_type: 'access',
  user_id: '65172aac6ebdfb77523a08d4',
  iat: 1696873680840,
  exp: 1696884480839
}
```



Ahora que comprobamos que nos esta devolviendo los datos procedemos a dejarlo de la siguiente manera:

```
controllers > js user.controller.js > <unknown>
1  const User = require("../models/user.model")
2
3  async function getMe(req, res) {
4
5      const { user_id } = req.user
6      const response = await User.findById(user_id)
7
8      if(!response){
9          res.status(400).send({msg: "No se ha encontrado el usuario"})
10     } else {
11         res.status(200).send(response)
12     }
13 }
14
15 module.exports = {
16     getMe,
17 }
```

Ahora verificamos de nuevo en nuestro insomnia al enviar la petición:



The screenshot shows the insomnia API client interface. A request is made to `/user/me` using the `GET` method. The response is `200 OK` with a size of `211 B`. The response body is a JSON object:

```
1 + {
2     "_id": "65172aac6ebdfb77523a08d4",
3     "firstname": "Jimmy",
4     "lastname": "Lombana",
5     "email": "jimmy@gmail.com",
6     "role": "user",
7     "active": true,
8     "password": "$2a$10$VLT9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMrNib./CEHD7HtgMppG",
9     "__v": 0
10 }
```

Y podemos observar que nos esta devolviendo los datos del usuario, para verificar de nuevo podemos probar con otro usuario de los registrados recordemos que tenemos en nuestra DB más:



The screenshot shows the insomnia API client interface with the results of a query. The results are labeled `QUERY RESULTS: 1-2 OF 2`. Two user documents are listed:

```
_id: ObjectId('65172ace26ebdfb77523a08d8')
firstname: "Cristian"
lastname: "Lobaton"
email: "elpinotor@gmail.com"
role: "user"
active: false
password: "$2a$10$FVOocCGleAAvdcm8m8PK9.PNxvsfstpkL3nIkaQ8J6vpjOhjFIIW"
__v: 0

_id: ObjectId('65172aac6ebdfb77523a08d4')
firstname: "Jimmy"
lastname: "Lombana"
email: "jimmy@gmail.com"
role: "user"
active: true
password: "$2a$10$VLT9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMrNib./CEHD7HtgMppG"
__v: 0
```

Y para ello los enviamos a través de insomnia:



Y recordemos que el usuario que tengo no esta activo, así que lo voy a activar en mongo y vuelvo a enviar la petición:

```
_id: ObjectId('65172ce26ebdfb77523a08d8')
firstname: "Cristian"
lastname: "Lobaton"
email: "elpintor@gmail.com"
role: "user"
active: true
password: "$2a$10$FV0ocCGlEAvgcWmm8PK9.PNvxfstpkL3nIkaQ8J6vpj0hjFIIW"
__v: 0
```

Aquí ya me devuelve los tokens así que vamos a tomar de nuevo el token y lo ingresamos en insomnia con una nueva autorización:

Ahora tengo los datos de este usuario, y si comprobamos de nuevo con el otro usuario:

Me devuelve los datos de este usuario, de esta manera ya hemos validado que los tokens nos funcionen y validen la información de nuestro usuario.



OBTENER TODOS LOS USUARIOS

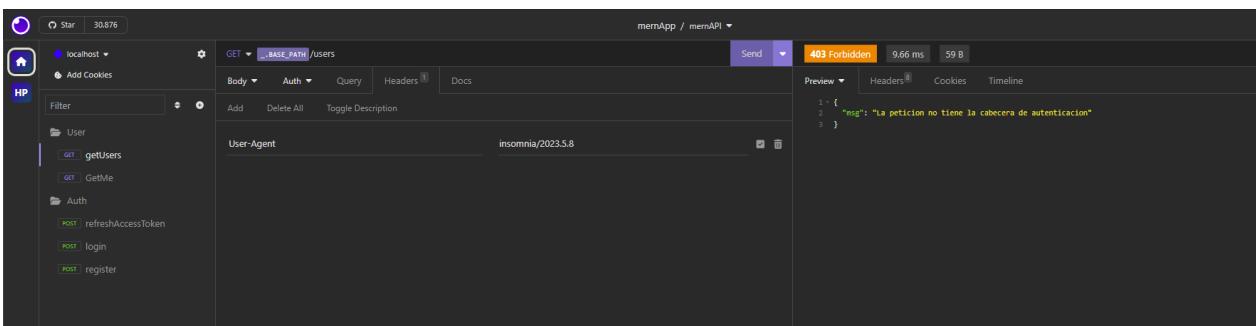
Vamos a crear un endpoint que nos devuelva todos los usuarios, con las condiciones de que nos muestre los activos y los no activos, para ello vamos a nuestro **user.controller.js**

```
controllers > user.controller.js > <unknown>
1 const User = require("../models/user.model")
2
3 async function getMe(req, res) {
4
5     const { user_id } = req.user
6     const response = await User.findById(user_id)
7
8     if(!response){
9         res.status(400).send({msg: "No se ha encontrado el usuario"})
10    } else {
11        res.status(200).send(response)
12    }
13}
14
15 async function getUsers(req, res){
16     res.status(200).send({msg: "Ok"})
17 }
18
19 module.exports = {
20     getMe,
21     getUsers,
22 }
```

Acá vamos primero a realizar una validación de nuestra ruta, para ello creamos **getUsers** y lo exportamos, y nos dirigimos a nuestro **user.router.js** y vamos a crear la ruta:

```
router > user.router.js > ...
1 const express = require("express")
2 const UserController = require("../controllers/user.controller")
3 const md_auth = require("../middlewares/authenticated")
4
5 const api = express.Router()
6
7 api.get("/user/me", [md_auth.ensureAuth], UserController.getMe)
8 api.get("/users", [md_auth.ensureAuth], UserController.getUsers)
9
10 module.exports = api
```

Ahora nos vamos a nuestro insomnia y vamos a crear una nueva petición que se llame **getUsers**:



Al momento de darle a la solicitud me dice que no tiene la cabecera de autenticación, quiere decir que está funcionando la validación de nuestro token, para ello recordemos que vamos a darle en headers y cargamos la información para la autorización:



De esta manera ya me devuelve el mensaje ok para indicar que la ruta esta validada, ahora vamos a aplicar la lógica de la función, lo primero es obtener el query de la función:

```
controllers > js user.controller.js >  <unknown>
14
15  async function getUsers(req, res){
16
17    const { active } = req.query
18    console.log("Active → ", active)
19
20    res.status(200).send({msg: "Ok"})
21
22 }
23
```

De esta manera vamos a obtener nuestro query en consola de los activos, ahora vamos a insomnia ejecutamos la petición y para poder verificarlo revisamos la consola:

```
[nodemon] 1.3.7 starting 'node index.js'
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
Active -> undefined
```

De esta manera nos muestra que no esta definido si son activos o no activos, para que nos muestre debemos indicarle a través de insomnia de la siguiente manera:

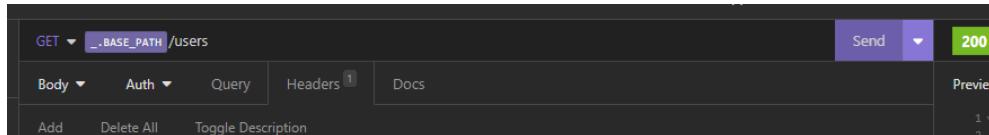
Y en consola nos muestra:

```
http://localhost:3000/api/v1
Active -> undefined
Active -> true
```

Y si colocamos false:

```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
Active -> undefined
Active -> true
Active -> false
|
```

Por el momento lo dejaremos así de esta manera:



Para poder manejarlos todos, así que ahora vamos a nuestro **user.controller.js**:

```
14
15  async function getUsers(req, res){
16
17      const { active } = req.query
18
19      let response = null
20
21      if (active === undefined){
22          response = await User.find()
23      } else {
24          response = await User.find({ active })
25      }
26
27      console.log(response);
28      res.status(200).send({msg: "Ok"})
29  }
30
31  module.exports = {
32      getMe,
33      getUsers,
34  }
```

Le pasamos en un **let response**, y mediante una condición le indicamos si es igual a **undefined**, y le decimos que nos devuelva los datos de estos, por medio de consola mostramos **response**:

```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
[
{
    _id: new ObjectId("65172aac6ebdfb77523a08d4"),
    firstname: 'Jimmy',
    lastname: 'Lombana',
    email: 'jimmy@gmail.com',
    role: 'user',
    active: true,
    password: '$2a$10$VLt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMUrNjb./CEHD7HtgMppG',
    __v: 0
},
{
    _id: new ObjectId("65172ce26ebdfb77523a08d8"),
    firstname: 'Cristian',
    lastname: 'Lobaton',
    email: 'elpintor@gmail.com',
    role: 'user',
    active: true,
    password: '$2a$10$FVOocCGIEAAvdclmm8PK9.PNxvsfstpkL3nIkaQ8J6vpj0h.jFIIW',
    __v: 0
}
]
```

Y acá podemos observar nuestros datos de usuarios, todos sin excepción, vamos a modificar a uno de los usuarios y lo dejaremos en estado active false:

```
QUERY RESULTS: 1-2 OF 2

_id: ObjectId('65172aac6ebdfb77523a08d4')
firstname: "Jimmy"
lastname: "Lombana"
email: "jimmy@gmail.com"
role: "user"
active: false
password: "$2a$10$Vlt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG"
__v: 0
```

Y volvemos a ejecutar cambiando la petición de insomnia por true:



GET `BASE_PATH` /users?active=true

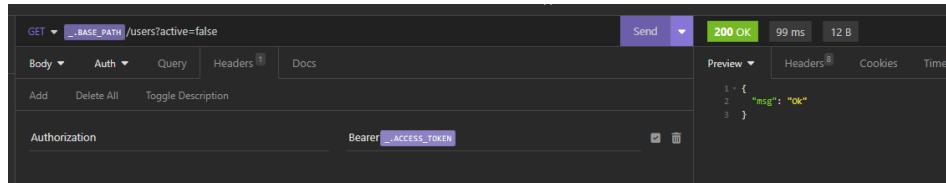
Send **200 OK** 97.4 ms 12 B

Preview `{ "msg": "ok" }`

Y en consola nos mostrara:

```
http://localhost:3000/api/v1
[
  {
    _id: new ObjectId("65172ce26ebdfb77523a08d8"),
    firstname: 'Cristian',
    lastname: 'Lobaton',
    email: 'elpinfor@gmail.com',
    role: 'user',
    active: true,
    password: '$2a$10$FVDocCGLEAvdcUmm8PK9.PNxvsfstpkL3UnIkaQ8J6vp.j0hJFIU',
    __v: 0
  }
]
```

Y si la dejamos en false:



GET `BASE_PATH` /users?active=false

Send **200 OK** 99 ms 12 B

Preview `{ "msg": "ok" }`

Nos mostrara lo siguiente:

```
[
  {
    _id: new ObjectId("65172aac6ebdfb77523a08d4"),
    firstname: 'Jimmy',
    lastname: 'Lombana',
    email: 'jimmy@gmail.com',
    role: 'user',
    active: false,
    password: '$2a$10$Vlt9i8VW.CQe.ekAnZC.W.tKChFSGBDIQMWrNib./CEHD7HtgMppG',
    __v: 0
  }
]
```

De esta manera ya tenemos el control sobre nuestra data, por último, eliminamos el log con el que mostramos la información y modificamos nuestro status 200 con response:



```

controllers > js user.controller.js > ...
14
15     async function getUsers(req, res){
16
17         const { active } = req.query
18
19         let response = null
20
21         if (active === undefined){
22             response = await User.find()
23         } else {
24             response = await User.find({ active })
25         }
26
27         res.status(200).send(response)
28
29     }
30
31     module.exports = {
32         getMe,
33         getUsers,
34     }

```

De esta manera nos devolverá en nuestro insomnia la data:

```

GET <BASE_PATH> /users
Send ▾ 200 OK | 114 ms | 432 B
Body ▾ Auth ▾ Query Headers Docs
Add Delete All Toggle Description
Authorization: Bearer 
Preview ▾ Headers Cookies Timeline
1: [
2:   {
3:     "_id": "65172aaac6ebdfb77523a08d4",
4:     "firstname": "jimmy",
5:     "lastname": "lombana",
6:     "email": "jimmy@gmail.com",
7:     "role": "user",
8:     "active": false,
9:     "password": "$2a$10$V0ocCG1EAwdCMmBPK9.PNxvsfstpkL3UnIkaQ8J6vpjOhjF1IM",
10:    "__v": 0
11  },
12  {
13    "_id": "65172ce5ebdfb77523a08d8",
14    "firstname": "cristian",
15    "lastname": "lobaton",
16    "email": "elpintor@gmail.com",
17    "role": "user",
18    "active": true,
19    "password": "$2a$10$V0ocCG1EAwdCMmBPK9.PNxvsfstpkL3UnIkaQ8J6vpjOhjF1IM",
20    "__v": 0
21  }
22 ]

```

CREANDO USUARIOS

Ahora vamos a crear nuestros usuarios desde el panel de administrador para ello vamos a crear la función **createUser** dentro de **user.controller.js**:

```

controllers > js user.controller.js >  <unknown>
30
31     async function createUser(req, res){
32         res.status(200).send({msg: "Funciona!"})
33     }
34
35     module.exports = {
36         getMe,
37         getUsers,
38          createUser,
39     }

```

Acá dejamos un mensaje de estatus 200 para verificar que funciona, ahora nos vamos a nuestras rutas **user.router.js** y allí vamos a crear nuestro endpoint:



```

router > user.router.js > ...
1  const express = require("express")
2  const UserController = require("../controllers/user.controller")
3  const md_auth = require("../middlewares/authenticated")
4
5  const api = express.Router()
6
7  api.get("/user/me", [md_auth.asureAuth], UserController.getMe)
8  api.get("/users", [md_auth.asureAuth], UserController.getUsers)
9  api.post("/user", [md_auth.asureAuth], UserController.createUser)
10 |
11 module.exports = api

```

Ahora que lo creamos vamos a insomnia y vamos a crear una nueva petición tipo **POST** que será **createUser** a través de **/user**

The screenshot shows the Insomnia REST Client interface. On the left, there's a sidebar with a tree view of API endpoints under 'User' and 'Auth'. In the main area, a POST request to '/user' is selected. The status bar at the top right shows '403 Forbidden' with a response time of '17.3 ms' and a size of '59 B'. Below the status, there's a 'Preview' tab showing the JSON response: { "msg": "La petición no tiene la cabecera de autenticación" }. Other tabs include 'Headers', 'Cookies', and 'Timeline'.

Cuando la ejecutamos nos dice que no tenemos la cabecera de **auth**, para ello debemos agregar la validación de la petición y colocamos nuestro token:

This screenshot shows the same Insomnia interface after adding an Authorization header with a token. The status bar now shows '200 OK' with a response time of '12.3 ms' and a size of '19 B'. The 'Preview' tab shows the JSON response: { "msg": "Funcional" }. The 'Headers' tab shows the added 'Authorization' header with the value 'Bearer ACCESS_TOKEN'.

De esta manera ya nos funciona, ahora realizaremos un log de nuestro req.body:

```

controllers > user.controller.js > ...
30
31  async function createUser(req, res){
32    console.log(req.body)
33    res.status(200).send({msg: "Funciona!"})
34  }
35
36  module.exports = {
37    getMe,
38    getUsers,
39    createUser,
40  }

```

Vamos a insomnia en donde realizaremos de nuevo la petición y en consola obtendremos lo siguiente:

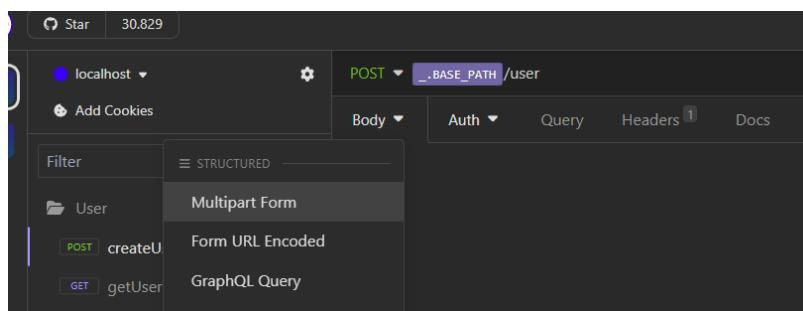


```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
#####
### MERN API REST #####
#####
http://localhost:3000/api/v1
{}
```

Nos llega un objeto vacío, pero acá no debería llegar vacío, nos deben pasar los datos de nuestro usuario, recordemos que los datos de nuestro modelo son los siguientes:

```
3 const UserSchema = mongoose.Schema({
4   firstname: {
5     type: String,
6     required: true,
7     trim: true
8   },
9   lastname: {
10    type: String,
11    required: true,
12    trim: true
13 },
14   email: {
15     type: String,
16     unique: true,
17     required: true
18 },
19   password: {
20     type: String,
21     required: true
22 },
23   role: String,
24   active: Boolean,
25   avatar: String,
26 })
```

Y estos datos son los que nos van a llegar a través de esta petición, y en especial recordemos que no todos son strings, tambien tendremos uno que será un fichero de imagen como lo es avatar, en insomnia este tipo de datos se enviaran a través de multipart, para ello vamos a configurarlo:



Y dejamos los datos que vamos a enviar a través de multipart de la siguiente manera:



Un multipart es un tipo de formulario que se utiliza para enviar datos binarios, como archivos, a un servidor web. Los datos binarios se dividen en partes más pequeñas, cada una con un encabezado que especifica el tipo de contenido y el nombre del campo, al momento de pasar el valor de nuestro avatar podemos seleccionar como tipo archivo:

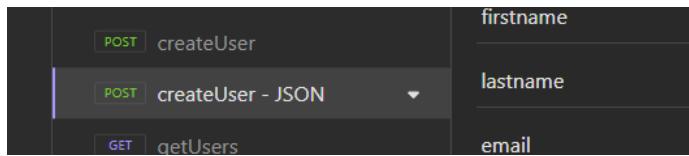
Por el momento lo dejaremos sin cargar un archivo, procedemos a enviar la solicitud y obtendremos que nos devuelve nuevamente un objeto vacío:

```
# This file is part of the MERN stack project.
# #####
# *** MERN API REST ***
# #####
http://localhost:3000/api/v1
{ }
{ }
```

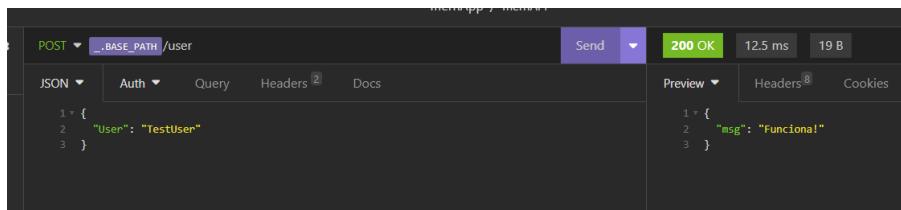
Si lo enviáramos a través de un JSON esto sería diferente, vamos a hacer la prueba de la siguiente manera, vamos a duplicar nuestra solicitud primero:

Y le colocaremos createUser – JSON





Y ahora en lugar de enviarlo como multipart lo enviaremos como JSON:



Y en consola obtendríamos lo siguiente:

```
#####
http://localhost:3000/api/v1
{}
{}
{ User: 'TestUser' }
```

Nos llega perfectamente los datos, pero cuando utilizamos **multipart** cambia el modo en el cual yo envío los datos y en especial si vamos a manejar imágenes, acá debemos utilizar un middleware que nos permita realizarlo y para ello vamos a utilizar uno que se llama **connect-multiparty**:

Y para ello procedemos a instalarlo en nuestro terminal ejecutando **yarn add connect-multiparty**

```
LAPTOPHDMI ~ D:\DATA\...\server yarn 18.16.0 2ms
{ hdtoledo } yarn add connect-multiparty
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
```

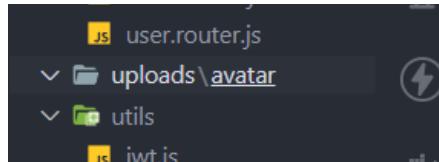
Vamos a levantar de nuevo nuestro servidor y procedemos a implementarlo en **user.router.js**

```
router > js user.router.js > multiparty
1 const express = require("express")
2 const multiparty = require("connect-multiparty")
3 const UserController = require("../controllers/user.controller")
4 const md_auth = require("../middlewares/authenticated")
```

Y ahora vamos a realizar lo siguiente allí mismo:

```
router > js user.router.js > md_upload
  5
  6  const md_upload = multiparty({ uploadDir: "./uploads/avatar" })
  7  const api = express.Router()
  8
```

Acá estamos definiendo nuestra const del middleware y utilizamos multiparty para indicarle donde queremos subir el avatar de nuestro usuario, recordemos que la ubicación debe estar creada así que vamos a crear la carpeta avatar dentro de **uploads**:



Y por último en nuestra ruta le pasamos el middleware **md_upload** que acabamos de implementar:

```
router > js user.router.js > ...
  8
  9  api.get("/user/me", [md_auth.asureAuth], UserController.getMe)
 10 api.get("/users", [md_auth.asureAuth], UserController.getUsers)
 11 api.post("/user", [md_auth.asureAuth, md_upload], UserController.createUser)
 12
 13 module.exports = api
```

Acá nosotros podemos pasarle cualquier cantidad de middlewares a nuestra ruta y no habrá ningún inconveniente, ahora probamos de nuevo ejecutando la petición a través del multipart en insomnia y en consola observaremos:

```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
{
  firstname: 'name01',
  lastname: 'lastname02',
  email: 'test01@gmail.com',
  password: '123456',
  role: 'admin',
  avatar: ''
}
```

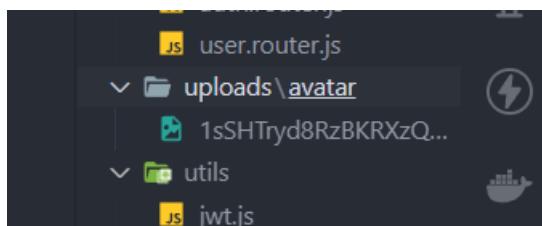
De esta manera hemos logrado pasar los datos a través del middleware y procesar el multipart, ahora probaremos con una imagen para revisar que funciona el cargue a través de insomnia:



Una vez cargada la imagen y enviada la solicitud nos fijamos en consola que nos saldrá lo siguiente:

```
[nodemon] starting `node index.js`
#####
### MERN API REST #####
#####
http://localhost:3000/api/v1
{
  firstname: 'name01',
  lastname: 'lastname02',
  email: 'test01@gmail.com',
  password: '123456',
  role: 'admin'
}
```

Se pasan los datos perfectamente y si revisamos nuestra carpeta de **uploads/avatar** observaremos que ya tenemos nuestra imagen cargada:



Nuestra imagen nos indica que fue procesada correctamente y se le cambia el nombre por seguridad y la almacena correctamente en nuestra carpeta, ahora lo que sigue es configurar nuestra función **createUser**, para ello vamos a realizar lo siguiente dentro de **user.controller.js**, lo primero será observar que el formato de nuestra contraseña nos llega en string y para ello debemos dejarla encriptada así que vamos a importar **bcrypt**

```
controllers > user.controller.js > bcrypt
1 const bcrypt = require("bcryptjs")
2 const User = require("../models/user.model")
3
```

Y ahora procedemos a dejar nuestra función de la siguiente manera:



```

controllers > user.controller.js > createUser
31
32  async function createUser(req, res){
33
34      const { password } = req.body
35      const salt = bcrypt.genSaltSync(10)
36      const hashPassword = bcrypt.hashSync(password, salt)
37
38      console.log(hashPassword)
39
40      res.status(200).send({msg: "Funciona!"})
41
42

```

Dentro de la función pasamos nuestra contraseña, aplicamos bcrypt para encriptarla y utilizamos el log para revisar que este pasando encriptada, en insomnia ejecutamos nuevamente la solicitud y deshabilitamos la imagen para que no nos esté guardando más esta:

Y en nuestro log nos muestra lo siguiente:

```

[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
$2a$10$4AZ7gfrH/xYVs6sddVvuJeB7L.W2myZqcW.y13Bj0K7gyGZPRBSCm
|

```

Ya tenemos encriptada nuestra contraseña, ahora seguiremos con la funcionalidad:

```

controllers > user.controller.js > createUser
32  async function createUser(req, res){
33
34      const { password } = req.body
35      const salt = bcrypt.genSaltSync(10)
36      const hashPassword = bcrypt.hashSync(password, salt)
37
38      const user = new User({ ...req.body, active: false, password: hashPassword })
39      console.log(user)
40
41      res.status(200).send({msg: "Funciona!"})
42
43

```

En la siguiente linea `const user = new User({ ...req.body, active: false, password: hashPassword })` crea una nueva instancia del modelo User, estableciendo la propiedad active en false y la propiedad password en la contraseña encriptada.



@hdtoledo

El operador de propagación `...req.body` copia todas las propiedades del objeto `req.body` en el nuevo objeto `user`. Esta es una forma conveniente de poblar el objeto `User` con los datos que se enviaron en el cuerpo de la solicitud.

La función `hashPassword` es responsable de encriptar la contraseña antes de almacenarla en la base de datos. Esta es una medida de seguridad importante para evitar que los atacantes accedan a las contraseñas de los usuarios en caso de una violación de datos.

Y ahora volvemos a enviar la solicitud de nuevo a través de insomnia y al verla reflejada en consola obtendremos los datos con nuestra contraseña ya encriptada:

```
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
{
  firstname: 'name01',
  lastname: 'lastname02',
  email: 'test01@gmail.com',
  password: '$2a$10$t4DlexXoTdFX/e4VpJ.txTnbfh/vFarSKqdv3H1rbQ0aW/oMtu',
  role: 'admin',
  active: false,
  _id: new ObjectId("6526e107c843bbe21cb8490e")
}
```

Vamos a realizar una pequeña modificación en la estructura del código y agregar nuestro avatar:

```
controllers > user.controller.js > createUser
32  async function createUser(req, res){
33
34    const { password } = req.body
35    const user = new User({ ...req.body, active: false })
36
37    const salt = bcrypt.genSaltSync(10)
38    const hashPassword = bcrypt.hashSync(password, salt)
39
40    user.password = hashPassword
41
42    console.log(user)
43
44    res.status(200).send({msg: "Funciona!"})
45 }
```

La modificación dentro de este código fue simplemente subir nuestra const `user`, eliminamos `password` de allí y la dejamos mas abajo, esto no altera nuestra funcionalidad si revisamos en consola saldrá igual:

```
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
{
  firstname: 'name01',
  lastname: 'lastname02',
  email: 'test01@gmail.com',
  password: '$2a$10$t4DlexXoTdFX/e4VpJ.txTnbfh/vFarSKqdv3H1rbQ0aW/oMtu',
  role: 'admin',
  active: false,
  _id: new ObjectId("6526e2e376d9c330cf56f50e")
```



@hdtoledo

Ahora procedemos a dejar lista nuestra imagen, recordemos que el usuario no esta obligado a subir un avatar, vamos a revisar como manipular la data del avatar, mediante un logo realizamos lo siguiente:

```
39     user.password = hashPassword
40
41
42     console.log(req.files.avatar)
43
44     res.status(200).send({msg: "Funciona!"})
45 }
```

En consola obtendremos esto:

```
#####
### MERN API REST #####
#####
http://localhost:3000/api/v1
{
  fieldName: 'avatar',
  originalFilename: 'logo512.png',
  path: 'uploads\avatar\ge56QvkLYaBgksUVDgKuTiCO.png',
  headers: {
    'content-disposition': 'form-data; name="avatar"; filename="logo512.png"',
    'content-type': 'image/png'
  },
  size: 27548,
  name: 'logo512.png',
  type: 'image/png'
}
```

Todo esto es la data que debemos procesar de nuestra imagen, por ahora dejaremos un pendiente allí ya que lo realizaremos de otra manera asi que vamos a dejarlo de la siguiente manera:

```
39     user.password = hashPassword
40
41
42     if(req.files.avatar){
43         //TODO:
44         console.log("Procesar avatar")
45     }
46
47     res.status(200).send({msg: "Funciona!"})
48 }
```

Y vamos a dejar el guardado de nuestros datos de la siguiente manera:

```
controllers > user.controller.js > ...
41
42     if(req.files.avatar){
43         //TODO:
44         console.log("Procesar avatar")
45     }
46
47     user.save((error, userStored) => {
48         if (error){
49             res.status(400).send({msg: "Error al crear el usuario !"})
50         } else {
51             res.status(201).send(userStored)
52         }
53     })
54
55     res.status(200).send({msg: "Funciona!"})
56 }
57 }
```

Vamos a ejecutar de nuevo en insomnia la petición y al realizarla observaremos:



```

POST _/BASE_PATH/_User
Send 201 Created 231 ms 218 B Just Now
Preview Headers Cookies Timeline
1 * {
2   "firstname": "name01",
3   "lastname": "lastname02",
4   "email": "test01@gmail.com",
5   "password": "$2a$05$fr3/fUpvjlIcl4Xjk.P1l0u.SUr7lvQss9Nr9EBI3EdkZzciW4W",
6   "role": "admin",
7   "active": false,
8   "_id": "6526e706964b3efffd9c6f9c",
9   "_v": 0
10 }

```

De esta manera observamos que nos devuelve los datos en mensaje satisfactorio, y al revisar en **mongodb** vamos a ver como se nos creó un tercer usuario:

```

test.users
STORAGE SIZE: 36KB LOGICAL DATA SIZE: 0B TOTAL DOCUMENTS: 3 INDEXES TOTAL SIZE: 72KB
Find Indexes Schema Anti-Patterns Aggregation Search Indexes
INSERT DOCUMENT
Filter Type a query: { field: 'value' }
Reset Apply More Options ▾
QUERY RESULTS: 1-3 OF 3
_id: ObjectId('65172aac6ebdfb77523a88d4')
firstname: "Immy"
lastname: "Lebanan"
email: "Immy@gmail.com"
role: "admin"
active: false
password: "$2a$10$VLTs18W.W.CQo.ekAnZC.W.tKHfSGBDIQMrN1b./CEHDYItgMppG"
__v: 0

_id: ObjectId('65172ce26ebdfb77523a88d8')
firstname: "Cristian"
lastname: "Lobaton"
email: "elspinotor@gmail.com"
role: "user"
active: true
password: "$2a$10$FVOocCG1EAAvdcLmmPK9.PNxvsfstpkL3UlnIkaQ36vpj0hJFIW6"
__v: 0

_id: ObjectId('6526e706964b3efffd9c6f9c')
firstname: "name01"
lastname: "lastname02"
email: "test01@gmail.com"
password: "$2a$10$fr3/fUpvjlIcl4Xjk.P1l0u.SUr7lvQss9Nr9EBI3EdkZzciW4W"
role: "admin"
active: false
__v: 0

```

Y al volver a intentar enviar los mismos datos en insomnia vamos a obtener el error al crear el usuario:

```

POST _/BASE_PATH/_User
Send 400 Bad Request 183 ms 37 B
Preview Headers Cookies Timeline
1 * {
2   "msg": "Error al crear el usuario !"
3 }

```

Y si cambiamos el correo electrónico y volvemos a enviarlo, notaremos que nos crea el usuario, ya que como hemos dejado requerido el correo pues no tendremos errores al momento de crearlos:



```

1 + {
2   "firstname": "Probando",
3   "lastname": "lastname02",
4   "email": "test02@gmail.com",
5   "password": "$2a$10$803POg2epbQf7gxYJPa/..ZNgnx0Q6Ir9G47RusTeLM7DNYZkq6u",
6   "role": "admin",
7   "active": false,
8   "_id": "6526e8bf964b3efffd9c6fa0",
9   "__v": 0
10  }

```

Y en mongodb:

```

{
  "_id": "6526e8bf964b3efffd9c6f9c",
  "firstname": "name01",
  "lastname": "lastb0n",
  "email": "elpintor@gmail.com",
  "password": "$2a$10$FV0ocCG1EAAvdcMm8PK9.PNxvsfstpkL3UnIkaQsJ6vpj0hjFIW",
  "role": "user",
  "active": true,
  "__v": 0
}

{
  "_id": "6526e8bf964b3efffd9c6fa0",
  "firstname": "name01",
  "lastname": "lastb0n",
  "email": "test02@gmail.com",
  "password": "$2a$10$7r3/fUpvjiIc4Xjk.P110u.5Ur7lvQ5s9iNr9EB13EdkZ2cIW4W",
  "role": "admin",
  "active": false,
  "__v": 0
}

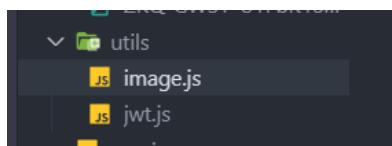
{
  "_id": "6526e8bf964b3efffd9c6fa0",
  "firstname": "Probando",
  "lastname": "lastname02",
  "email": "test02@gmail.com",
  "password": "$2a$10$803POg2epbQf7gxYJPa/..ZNgnx0Q6Ir9G47RusTeLM7DNYZkq6u",
  "role": "admin",
  "active": false,
  "__v": 0
}

```

Así de esta manera ya tenemos el modulo de creación de usuario desde nuestro panel de administrador y este lo conectaremos cuando estemos con el front.

PROCESANDO LA IMAGEN

Ahora vamos a procesar la imagen y debemos definirla y setearla, para ello vamos a crear una función para poderla procesar, para ello nos vamos a **utils** y vamos a crear **image.js**:



Y dentro vamos a colocar lo siguiente:



```
utils > ls image.js > ...
1  function getFilePath(file) {
2    const filePath = file.path
3
4    return filePath
5  }
6
7  module.exports = {
8    getFilePath,
9  }
```

En esta función que vamos a empezar a estructurar primero almacenamos la ruta de nuestra imagen y nos mostrara el path, ahora vamos a **user.controller.js** en donde vamos a importarlo:

```
controllers > ls user.controller.js > image
1  const bcrypt = require("bcryptjs")
2  const User = require("../models/user.model")
3  const image = require("../utils/image")
4
```

Y ahora en nuestra función va a quedar de la siguiente manera:

```
controllers > ls user.controller.js > createUser
41
42  user.password = hashPassword
43
44  if (req.files.avatar){
45    const imagePath = image.getFilePath(req.files.avatar)
46    console.log(imagePath)
47
48  user.save((error, userStored) => {
```

Y ahora vamos a nuestro insomnia y ejecutamos nuevamente la petición:

```
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
uploads\avatar\xBcdhyttRJqeGTu9Y-w_fXSy.png
|
```

Justo debajo obtenemos la ruta de la imagen, ahora vamos a limpiar nuestra ruta para almacenarla correctamente en un string, lo primero que vamos a realizar es lo siguiente:

```
utils > ls image.js > ...
1  function getFilePath(file) {
2    const filePath = file.path
3    const fileSplit = filePath.split("\\\\")
4
5    return fileSplit
6  }
7
8  module.exports = {
9    getFilePath,
10 }
```

En donde vamos a eliminar nuestro “/” si es en mac o “\\\\” si es en Windows, ahora volvemos a darle en insomnia para verificar y en consola nos saldrá lo siguiente:



@hdtolledo

```
#####
### MERN API REST #####
#####
http://localhost:3000/api/v1
[ 'uploads', 'avatar', 'Mm-7kjODyMe1CtRago8XHJw3.png' ]
```

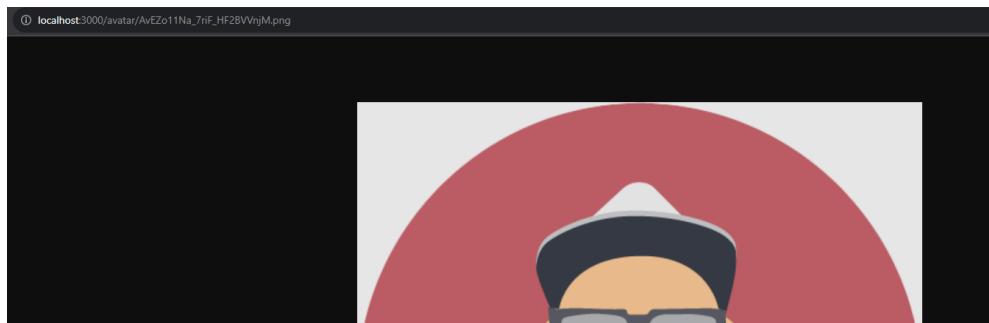
Un array con los datos del path, ahora vamos a armar la ruta correcta:

```
utils > js imagejs > ...
1 function getFilePath(file) {
2   const filePath = file.path
3   const fileSplit = filePath.split("\\\\")
4
5   return `${fileSplit[1]}/${fileSplit[2]}`
6 }
7
8 module.exports = {
9   getFilePath,
10 }
```

En nuestro return utilizamos backticks y colocamos la posición de nuestro array que nos interesa que seria la 1 y 2, volvemos a ejecutar insomnia y en consola obtenemos lo siguiente:

```
[nodemon] starting `node index.js`
#####
### MERN API REST #####
#####
http://localhost:3000/api/v1
avatar/AvEZo11Na_7riF_HF2BVn.jM.png
```

Ahora si copiamos esta dirección de enlace y la colocamos en nuestro navegador nos debe devolver como resultado la imagen del usuario:



Ahora lo que nos queda es almacenarlo directamente sobre la data que vamos a enviar de User, para ello hacemos que **user.avatar** sea igual a **imagePath**:

```
controllers > js user.controller.js > ⚡ createUser
...
41     user.password = hashPassword
42
43     if (req.files.avatar){
44       const imagePath = image.getPath(req.files.avatar)
45       user.avatar = imagePath
46
47
48     user.save((error, userStored) => {
```

Vamos a comprobar con un nuevo usuario y hacemos el envío a través de insomnia:

```
1 + {
2   "firstname": "Probando 2",
3   "lastname": "lastname03",
4   "email": "test04@gmail.com",
5   "password": "$2a$10$Komi9YwJW9XjntsZ4m20.zOZprr8GrvjPJ2M9Vkg8gSMxJTTvaa",
6   "role": "admin",
7   "active": false,
8   "_id": "6527005960b8ad666ca60ff",
9   "avatar": "avatar/p8Tj0pyG8782-ULfyJndZRM.png",
10  "_v": 0
11 }
```

ahora si nos fijamos en la data enviada ya tenemos nuestra ruta del avatar.

ACTUALIZANDO EL USUARIO

Ahora vamos a proceder a crear la función para poder actualizar los datos del usuario y para ello vamos a crear dentro de nuestro **user.controller.js** la función **updateUser**

```
57  async function updateUser(req, res) {
58    const { id } = req.params
59    const userData = req.body
60
61    //Password
62    //Avatar
63
64    User.findByIdAndUpdate({ _id: id }, userData, (error) => {
65      if(error) {
66        res.status(400).send({msg: "Error al actualizar el usuario"})
67      } else {
68        res.status(200).send({msg: "Datos de usuario actualizados"})
69      }
70    })
71  }
72
73
74  module.exports = [
75    getMe,
76    getUsers,
77    createUser,
78    updateUser,
79  ]
```

La función comienza por obtener el identificador del usuario a actualizar de los parámetros de la solicitud. Luego, obtiene los datos del usuario del cuerpo de la solicitud. Si es así, la función actualiza los campos correspondientes en el documento del usuario.

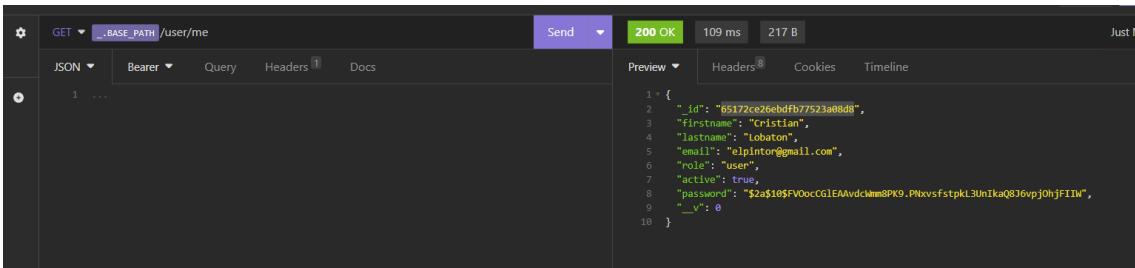
Finalmente, la función utiliza el método **findByIdAndUpdate** de Mongoose para actualizar los datos del usuario en la base de datos. Si la actualización se realiza correctamente, la función envía una respuesta al cliente con el código de estado 200 y el mensaje "Datos de usuario actualizados". Si se produce un error durante la actualización, la función envía una respuesta al cliente con el código de estado 400 y el mensaje "Error al actualizar el usuario".



Ahora vamos a crear el endpoint para nuestra función de actualización, nos vamos a **user.router.js** y vamos a colocar lo siguiente:

```
router > js user.router.js > ...
5
6  const md_upload = multiparty({ uploadDir: "./uploads/avatar" })
7  const api = express.Router()
8
9  api.get("/user/me", [md_auth.asureAuth], UserController.getMe)
10 api.get("/users", [md_auth.asureAuth], UserController.getUsers)
11 api.post("/user", [md_auth.asureAuth, md_upload], UserController.createUser)
12 api.patch("/user/:id", [md_auth.asureAuth, md_upload], UserController.updateUser)
13
14 module.exports = api
```

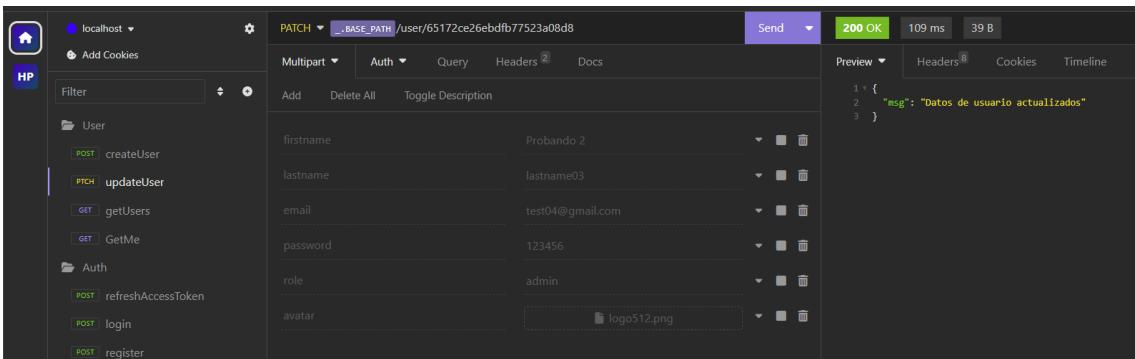
En esta ruta colocamos patch para poder realizar la actualización de los datos a través del id, implementamos los middlewares y cargamos la función de updateUser, para probarlo nos vamos a insomnia y vamos getMe para obtener nuestro id de usuario:



The screenshot shows the insomnia API client interface. A GET request is made to `/_BASE_PATH/user/me`. The response is 200 OK, took 109 ms, and is 217 B in size. The response body is a JSON object representing a user profile:

```
1: {
2:   "id": "65172ce26ebdbf77523a08d8",
3:   "firstname": "Cristian",
4:   "lastname": "Lobaton",
5:   "email": "elpinor@gmail.com",
6:   "role": "user",
7:   "active": true,
8:   "password": "$2a$10$FVOocCGlEAAvdcIwmSPK9.PNxvsfstpkL3UnIkaQ8J6vpjOHjFIIW",
9:   "_v": 0
10 }
```

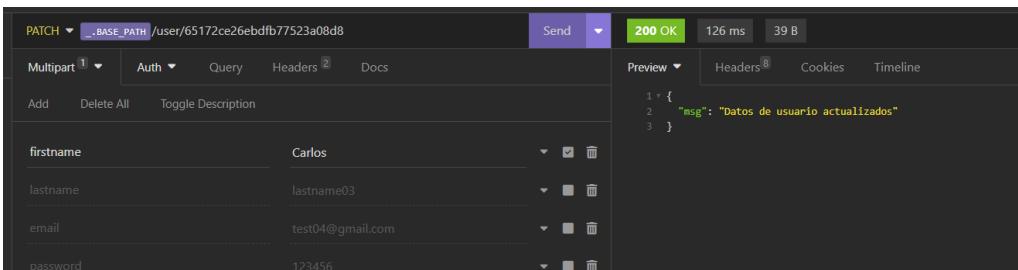
Y ahora vamos a crear una nueva petición y probamos el envío, desmarcamos todo solo para probar:



The screenshot shows the insomnia API client interface. A PATCH request is made to `/_BASE_PATH/user/65172ce26ebdbf77523a08d8`. The response is 200 OK, took 109 ms, and is 39 B in size. The response body is a JSON object with a message:

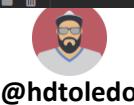
```
1: {
2:   "msg": "Datos de usuario actualizados"
3: }
```

Ahora que ya verificamos que si esta funcionando, vamos a probar a actualizar el nombre de usuario y para ello lo dejamos así:



The screenshot shows the insomnia API client interface. A PATCH request is made to `/_BASE_PATH/user/65172ce26ebdbf77523a08d8`. The response is 200 OK, took 126 ms, and is 39 B in size. The response body is a JSON object with a message:

```
1: {
2:   "msg": "Datos de usuario actualizados"
3: }
```



Para verificarlo podemos hacerlo en insomnia por medio de getMe enviamos la petición y obtenemos que si ha cambiado el nombre:

The screenshot shows the Insomnia REST Client interface. The URL is `GET /user/me`. The response code is **200 OK**, with a response time of 105 ms and a size of 215 B. The response body is a JSON object:

```
1: {  
2:   "_id": "65172ce26ebdfb77523a08d8",  
3:   "firstname": "Carlos",  
4:   "lastname": "Lobaton",  
5:   "email": "elpintor@gmail.com",  
6:   "role": "user",  
7:   "active": true,  
8:   "password": "$2a$10$FVOocCG1EAAvdchm8PK9.PNxvsfstpl3UnIkaQ8J6vpjOhjFIIM",  
9:   "__v": 0  
10: }
```

Y si verificamos en mongodb obtenemos que el usuario tambien ha sido actualizado:

The screenshot shows the MongoDB shell with the following query results:

```
QUERY RESULTS: 1-6 OF 6  
  
_id: ObjectId('65172ce26ebdfb77523a08d4')  
firstname: "Jimmy"  
lastname: "Lombana"  
email: "jimmy@gmail.com"  
role: "user"  
active: false  
password: "$2a$10$FVOocCG1EAAvdchm8PK9.PNxvsfstpl3UnIkaQ8J6vpjOhjFIIM"  
__v: 0  
  
  
_id: ObjectId('65172ce26ebdfb77523a08d8')  
firstname: "Carlos"  
lastname: "Lobaton"  
email: "elpintor@gmail.com"  
role: "user"  
active: true  
password: "$2a$10$FVOocCG1EAAvdchm8PK9.PNxvsfstpl3UnIkaQ8J6vpjOhjFIIM"  
__v: 0  
  
  
_id: ObjectId('6526a706964b3effd9c6f9c')
```

Ahora vamos a probar a cambiar los demás datos:

The screenshot shows the Insomnia REST Client interface. The URL is `PATCH /user/65172ce26ebdfb77523a08d8`. The response code is **200 OK**, with a response time of 109 ms and a size of 39 B. The response body is a JSON object:

```
1: {  
2:   "msg": "Datos de usuario actualizados"  
3: }
```

Validamos en getMe:

The screenshot shows the Insomnia REST Client interface. The URL is `GET /user/me`. The response code is **200 OK**, with a response time of 105 ms and a size of 226 B. The response body is a JSON object:

```
1: {  
2:   "_id": "65172ce26ebdfb77523a08d8",  
3:   "firstname": "Cristian Fernando",  
4:   "lastname": "Lobaton",  
5:   "email": "test04@gmail.com",  
6:   "password": "123456",  
7:   "role": "user",  
8:   "active": true,  
9:   "avatar": "logo512.png",  
10: }
```

Probamos de nuevo cambiando solo el rol:

PATCH `./BASE_PATH` /user/65172ce26ebdfb77523a08d8

Send ▾

200 OK | 100 ms | 39 B

Multipart 1 ▾ Auth ▾ Query Headers 2 Docs

Add Delete All Toggle Description

firstname: Cristian Fernando
lastname: Lobaton
email: test04@gmail.com
password: 123456
role: admin
avatar: logo512.png

Preview ▾ Headers 8 Cookies Timeline

```
1: {  
2:   "msg": "Datos de usuario actualizados"  
3: }
```

Y en getMe nos muestra:

GET `./BASE_PATH` /user/me

Send ▾

200 OK | 105 ms | 227 B

JSON ▾ Bearer ▾ Query Headers 1 Docs

1: ...

Preview ▾ Headers 8 Cookies Timeline

```
1: {  
2:   "_id": "65172ce26ebdfb77523a08d8",  
3:   "firstname": "Cristian Fernando",  
4:   "lastname": "Lobaton",  
5:   "email": "elpintor@gmail.com",  
6:   "role": "admin",  
7:   "active": true,  
8:   "password": "$2a$10$FVOocCG1EAvdchmm8PK9.PNkvsfstpkL3UnIkaQ8JevpjOhjFIIM",  
9:   "__v": 0  
10: }
```

Ahora vamos con la contraseña y el avatar, para ello vamos a hacer el envío de la contraseña por el momento así como esta para verificar que pasa:

PATCH `./BASE_PATH` /user/65172ce26ebdfb77523a08d8

Send ▾

200 OK | 107 ms | 39 B

Multipart 1 ▾ Auth ▾ Query Headers 2 Docs

Add Delete All Toggle Description

firstname: Cristian Fernando
lastname: Lobaton
email: test04@gmail.com
password: 123456
role: admin
avatar: logo512.png

Preview ▾ Headers 8 Cookies Timeline

```
1: {  
2:   "msg": "Datos de usuario actualizados"  
3: }
```

Vemos que la contraseña pasa sin problema, pero al revisar en **getMe** observamos que la contraseña ya no está encriptada:

Preview ▾ Headers 8 Cookies Timeline

```
1: {  
2:   "_id": "65172ce26ebdfb77523a08d8",  
3:   "firstname": "Cristian Fernando",  
4:   "lastname": "Lobaton",  
5:   "email": "elpintor@gmail.com",  
6:   "role": "admin",  
7:   "active": true,  
8:   "password": "123456",  
9:   "__v": 0  
10: }
```



@hdtoledo

Para ello vamos a realizar lo siguiente dentro de nuestra función **updateUser**:

```
controllers > user.controller.js > updateUser
  ...
57  async function updateUser(req, res) {
58    const { id } = req.params
59    const userData = req.body
60
61    if (userData.password) {
62      const salt = bcrypt.genSaltSync(10)
63      const hashPassword = bcrypt.hashSync(userData.password, salt)
64      userData.password = hashPassword
65    } else {
66      delete userData.password
67    }
68
69
70
71    //Avatar
72
```

Comprobamos si el usuario ha proporcionado una nueva contraseña. Si es así, el código genera una sal y hash, la contraseña con la sal. El hash de la contraseña se almacena en el campo password del objeto userData, y si el usuario no ha proporcionado una nueva contraseña, el código elimina el campo password del objeto userData. Esto se hace porque es importante no almacenar contraseñas sin hash en una base de datos.

Vamos a comprobar de nuevo que pasa con la contraseña:

The screenshot shows a POSTMAN interface with a PATCH request to `/_BASE_PATH/user/65172ce26ebdfb77523a08d8`. The request body is a Multipart form-data with fields: `firstname` (Cristian Fernando), `lastname` (Lobaton), `email` (test04@gmail.com), `password` (123456), `role` (admin), and `avatar` (a file named logo512.png). The response is a 200 OK status with a duration of 211 ms and a response size of 39 B. The preview shows the JSON response: `{ "msg": "Datos de usuario actualizados" }`.

Y al comprobarlo con `getMe`:

The screenshot shows a POSTMAN interface with a GET request to `getMe`. The response is a 200 OK status with a duration of 118 ms and a response size of 227 B. The preview shows the JSON response: `{ "_id": "65172ce26ebdfb77523a08d8", "firstname": "Cristian Fernando", "lastname": "Lobaton", "email": "elpintor@gmail.com", "role": "admin", "active": true, "password": "$2a$10$5tPCAXjPw8RrzuuXHrXFaeFtA/cWxwI/VK2Pz8MHHK2XxKPuEEURq", "__v": 0 }`.

Nos damos cuenta que ahora si esta encriptada nuestra contraseña.



@hdtolledo

Ahora vamos con la imagen o avatar de nuestro usuario, si comprobamos en insomnia:

The screenshot shows the Insomnia REST client interface. A PATCH request is being made to the endpoint `/_BASE_PATH/user/65172ce26ebdfb77523a08d8`. The request body contains the following fields:

Field	Value
firstname	Cristian Fernando
lastname	Lobaton
email	test04@gmail.com
password	123456
role	admin
avatar	logo512.png

The response status is **200 OK**, with a response time of 109 ms and a response size of 39 B. The response body is:

```
1 ▾ {  
2   "msg": "Datos de usuario actualizados"  
3 }
```

Me dice que se ha enviado la información, pero al realizar la petición en `getMe` observamos:

The screenshot shows the Insomnia REST client interface. The **Preview** tab is selected, displaying the response body of a `getMe` request. The response is a JSON object containing the user's information, including the newly uploaded avatar:

```
1 ▾ {  
2   "_id": "65172ce26ebdfb77523a08d8",  
3   "firstname": "Cristian Fernando",  
4   "lastname": "Lobaton",  
5   "email": "elpintor@gmail.com",  
6   "role": "admin",  
7   "active": true,  
8   "password": "$2a$10$5tPCAXjPW8RrzuuXHrXFAeFtA/cWxwI/VK2Pz8MHHK2XxKPuEEURq",  
9   "__v": 0  
10 }
```

Que no estamos gestionando la ubicación del avatar para ello vamos a realizar lo siguiente dentro de nuestra función `updateUser` y verificamos que se estén enviando los datos:

```
controllers > user.controller.js > updateUser  
65     } else {  
66       delete userData.password  
67     }  
68  
69     if (req.files.avatar) {  
70       console.log(req.files.avatar)  
71     }  
72  
73     User.findByIdAndUpdate({ _id: id }, userData, (error) => {  
74       if(error) {
```

Vamos primero mediante la condición si hay un archivo de avatar pues que nos muestre el log de este, realizamos en insomnia la petición y en consola nos saldrá:



```
#####
### MERN API REST #####
#####
http://localhost:3000/api/v1
{
  fieldName: 'avatar',
  originalFilename: 'logo512.png',
  path: 'uploads\avatar\6AGM8ijG9ju6oDiJddfHIG1U.png',
  headers: {
    'content-disposition': 'form-data; name="avatar"; filename="logo512.png"',
    'content-type': 'image/png'
  },
  size: 27548,
  name: 'logo512.png',
  type: 'image/png'
}
```

Verificamos que nos devuelve el objeto con sus datos, para ello vamos a realizar lo siguiente:

```
controllers > user.controller.js > updateUser
65     } else {
66       delete userData.password
67     }
68
69     if (req.files.avatar) {
70       const imagePath = image.getFilePath(req.files.avatar)
71       userData.avatar = imagePath
72     }
73
74   User.findByIdAndUpdate({ _id: id }, userData, (error) => {
75     if (error)
```

Agregamos al igual que lo hicimos con **imagePath** la ruta y se la pasamos a los datos de usuario, al comprobarlo de nuevo en insomnia por medio de **getMe** vamos a observar lo siguiente:

Preview		Headers	Cookies	Timeline
<pre>1 + { 2 "_id": "65172ce26ebdfb77523a08d8", 3 "firstname": "Cristian Fernando", 4 "lastname": "Lobato", 5 "email": "elpintor@gmail.com", 6 "role": "admin", 7 "active": true, 8 "password": "\$2a\$10\$5tPCAXjPw8RrzuiXirXFaeFTA/cbxwI/VK2Pz8MHK2XxKPuEEURq", 9 "__v": 0, 10 "avatar": "avatar/kAqX8gAGST1dx-Fji49xMkbQ.png" 11 }</pre>		8		

Ya obtenemos la ubicación de nuestro avatar y se pasa a los datos de nuestro usuario para ser almacenado en la base de datos.



@hdtoledo

ELIMINANDO USUARIO

Ahora realizaremos el ultimo endpoint que será la eliminación del usuario para ello vamos a crear la función **deleteUser** dentro de **user.controller.js**:

```
controllers > js user.controller.js > <unknown>
  ...
84  async function deleteUser(req, res){
85    const { id } = req.params
86
87    User.findByIdAndDelete(id, (error) => {
88      if(error){
89        res.status(400).send({msg: "Error al eliminar el usuario"})
90      } else {
91        res.status(200).send({msg: "Usuario Eliminado"})
92      }
93    })
94  }
95
96
97  module.exports = [
98    getMe,
99    getUsers,
100   createUser,
101  updateUser,
102  deleteUser,
103]
```

Realizamos nuestra función asíncrona para eliminar el registro a través del **id** del usuario, lo cargamos en una **const** como hicimos anteriormente y utilizamos la función para eliminar, colocamos los mensajes en caso de error y si se ejecuta normalmente, por último, exportamos la función, y ahora nos vamos para nuestro archivo de rutas **user.router.js** y allí vamos a crear la ruta:

```
router > js user.router.js > ...
  ...
5
6  const md_upload = multiparty({ uploadDir: "./uploads/avatar" })
7  const api = express.Router()
8
9  api.get("/user/me", [md_auth.asureAuth], UserController.getMe)
10 api.get("/users", [md_auth.asureAuth], UserController.getUsers)
11 api.post("/user", [md_auth.asureAuth, md_upload], UserController.createUser)
12 api.patch("/user/:id", [md_auth.asureAuth, md_upload], UserController.updateUser)
13 api.delete("/user/:id", [md_auth.asureAuth], UserController.deleteUser)
14
15
16  module.exports = api
```

Agregamos la ruta a través de delete y ejecutamos nuestro middleware de autenticación junto con la función de eliminar.

Procedemos a comprobarlo a través de insomnia, para ello vamos a consultar nuestros usuarios que se encuentran inactivos:





GET `_.BASE_PATH /users?active=false`

Send **200 OK** 97.1 ms 1145 B Just Now

Body Auth Query Headers Docs

Preview Headers Cookies Timeline

```

1 + [
2 + {
3 +   "_id": "65172aac6ebdfb77523a08d4",
4 +   "firstname": "Jimmy",
5 +   "lastname": "Lombana",
6 +   "email": "jimmy@gmail.com",
7 +   "role": "user",
8 +   "active": false,
9 +   "password": "$2a$10$VLt918W.CQe.ekAnZC.W.tKChFSGBDIQ%wRib./CEHD7HtgPppG",
10 +   "__v": 0
11 },
12 + {
13 +   "_id": "6526e706964b3efffd9c6f9c",
14 +   "firstname": "name01",
15 +   "lastname": "lastname02",
16 +   "email": "test0@gmail.com",
17 +   "password": "$2a$10$7rJUpvjl1cI4XJK.P1lOu.5Um71vQ5s9INh9EBI3Edk22cth4W",
18 +   "role": "admin",
19 +   "active": false,
20 +   "__v": 0
21 }
22 ]

```

Ya que obtenemos los usuarios, seleccionamos uno de los id y lo copiamos, nos dirigimos a crear un nuevo endpoint en insomnia que se llame deleteUser:



Star 30.839 mernApp / mernAPI

localhost Add Cookies

Filter User

- DELETE** `_.BASE_PATH /user/6526e706964b3efffd9c6f9c`
- Body** **Auth** **Query** **Headers** **Docs**
- Add Delete All Toggle Description
- Authorization Bearer `__ACCESS_TOKEN`

deleteUser createUser updateUser getUsers

Aplicamos el id en la ruta y nos disponemos a enviar la petición:



DELETE `_.BASE_PATH /user/6526e706964b3efffd9c6f9c`

Send **200 OK** 124 ms 27 B Just Now

Body Auth Query Headers Docs

Add Delete All Toggle Description

Authorization Bearer `__ACCESS_TOKEN`

Preview Headers Cookies Timeline

```

1 + {
2 +   "msg": "Usuario Eliminado"
3 }

```

Si todo va bien nos mostrara que se elimino el usuario y podemos disponernos a revisar mediante getUsers:



GET `_.BASE_PATH /users?active=false`

Send **200 OK** 106 ms 926 B Just Now

Body Auth Query Headers Docs

Add Delete All Toggle Description

Authorization Bearer `__ACCESS_TOKEN`

Preview Headers Cookies Timeline

```

1 + [
2 + {
3 +   "_id": "65172aac6ebdfb77523a08d4",
4 +   "firstname": "Jimmy",
5 +   "lastname": "Lombana",
6 +   "email": "jimmy@gmail.com",
7 +   "role": "user",
8 +   "active": false,
9 +   "password": "$2a$10$VLt918W.CQe.ekAnZC.W.tKChFSGBDIQ%wRib./CEHD7HtgPppG",
10 +   "__v": 0
11 },
12 + {
13 +   "_id": "6526e8bf964b3efffd9c6fa0",
14 +   "firstname": "Probando",
15 +   "lastname": "lastname02",
16 +   "email": "test02@gmail.com",
17 +   "password": "$2a$10$80JPog2ephQf7gxYJPa...ZNgnx0Q6ir9G47RusTelM7DihYZkqgu",
18 +   "role": "admin",
19 +   "active": false,
20 +   "__v": 0
21 }
22 + {
23 +   "_id": "6526e9393b71e8ea2b21321"
}

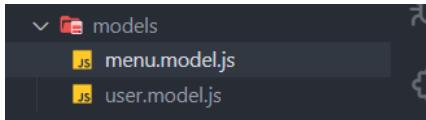
```

De esta manera terminamos de configurar nuestro CRUD de usuario.



MODELO MENU

El sistema de menú no se va quedará de manera estática, sino que se podrá activar y modificar directamente desde el panel del administrador, para ello vamos a crear un esquema que nos permita ser modificado directamente por el administrador, vamos a definirlo de la siguiente manera, creamos un archivo llamado **menu.model.js** dentro de **models**:



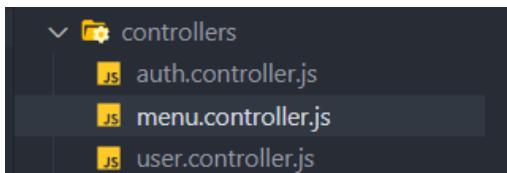
Y dentro vamos a definir nuestro menú de la siguiente manera:

```
models > JS menu.model.js > <unknown>
1  const mongoose = require("mongoose")
2
3  const MenuSchema = mongoose.Schema({
4      title: String,
5      path: String,
6      order: Number,
7      active: Boolean,
8  })
9
10 module.exports = mongoose.model("Menu", MenuSchema)
```

Nuestro modelo va a ser simple, ya que solo requerimos un título, una ruta, un orden y que se active o desactive.

ESTRUCTURA API MENU

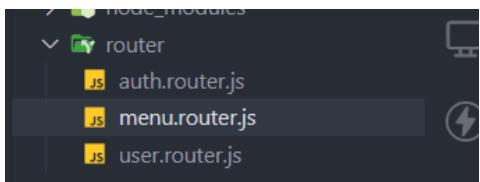
Ahora vamos a definir la estructura para el API menú, desde el controlador hasta exponerse al navegador, vamos a crear nuestro controlador **menu.controller.js** dentro de **controllers**:



Y dentro vamos a colocar toda la parte lógica para empezar a estructurar el menú:

```
controllers > JS menu.controller.js > ...
1  const Menu = require("../models/menu.model")
2
3  |
4
5  module.exports = {
6
7  }
```

Por el momento lo dejaremos así, ya que a continuación vamos a crear la ruta dentro de router nuestro **menu.router.js**:



Y dentro lo vamos a dejar de la siguiente manera:

```
router > js menu.router.js > ...
1 const express = require("express")
2 const MenuController = require("../controllers/menu.controller")
3 const md_auth = require("../middlewares/authenticated")
4
5 const api = express.Router()
6
7 //EndPoints
8
9
10 module.exports = api
```

Importamos express, nuestro controlador del menú, y nuestro middleware de autenticación y por último exportamos el módulo.

Ahora vamos a **app.js** y hacemos la importación de nuestro modulo:

```
js app.js > menuRoutes
9 // Importar rutas
10 const authRoutes = require("./router/auth.router")
11 const userRoutes = require("./router/user.router")
12 const menuRoutes = require("./router/menu.router")
13
14 //Configurar Body Parse
```

Y mas abajo colocamos la configuración de la ruta de la siguiente manera:

```
js app.js > ...
25
26 // Configurar Rutas
27 app.use(`/api/${API_VERSION}`, authRoutes)
28 app.use(`/api/${API_VERSION}`, userRoutes)
29 app.use(`/api/${API_VERSION}`, menuRoutes)
30
31
32 module.exports = app
```



@hdtoledo

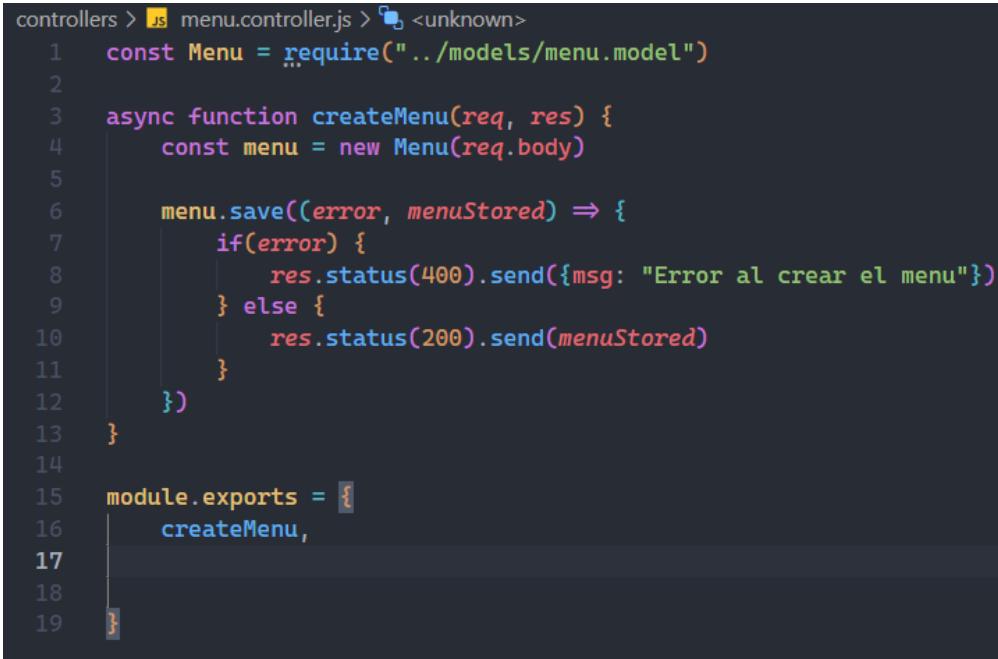
Ahora por último levantamos nuestro servidor y no debe generarnos ningún error:



```
LAPTOPHDMI > D:\DATA\...\server > yarn dev
{ @hdtledo } yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
```

CREANDO EL MENU

Ahora vamos a empezar con el primer menú así que nos ubicamos dentro de **menu.controller.js** y creamos nuestra función **createMenu**:



```
controllers > js menu.controller.js >  <unknown>
1 const Menu = require("../models/menu.model")
2
3 async function createMenu(req, res) {
4     const menu = new Menu(req.body)
5
6     menu.save((error, menuStored) => {
7         if(error) {
8             res.status(400).send({msg: "Error al crear el menu"})
9         } else {
10             res.status(200).send(menuStored)
11         }
12     })
13 }
14
15 module.exports = {
16     createMenu,
17
18 }
```

Creamos la función asíncrona llamada **createMenu()**, que se utiliza para crear un nuevo menú en una base de datos. La función recibe dos parámetros: **req** y **res**, que representan la solicitud y la respuesta **HTTP**, respectivamente.

El primer paso que realiza la función es crear un nuevo objeto **Menu** a partir del cuerpo de la solicitud **HTTP**. El cuerpo de la solicitud contiene los datos del nuevo menú, como su nombre, descripción, etc.

Una vez creado el objeto **Menu**, la función llama al método **save()** para guardarlo en la base de datos. El método **save()** es asíncrono, lo que significa que la función **createMenu()** no esperará a que se complete antes de devolver una respuesta.



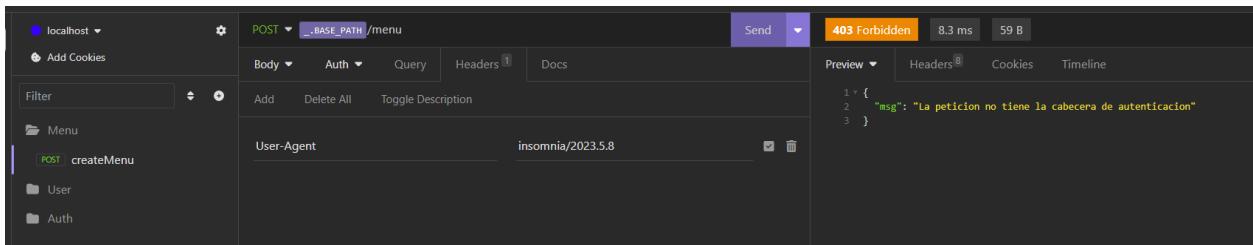
@hdtledo

Si el menú se guarda correctamente, la función `createMenu()` devolverá una respuesta HTTP con el estado 200 OK y el menú guardado. Si se produce un error al guardar el menú, la función devolverá una respuesta HTTP con el estado 400 Bad Request y un mensaje de error.

Ahora vamos a `menu.router.js` y creamos nuestro primer endpoint:

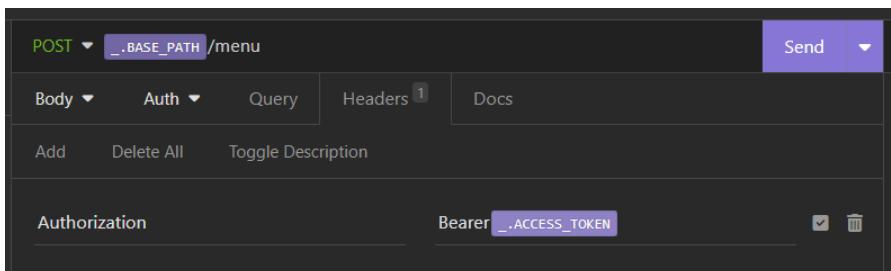
```
router > js menu.router.js > ...
1 const express = require("express")
2 const MenuController = require("../controllers/menu.controller")
3 const md_auth = require("../middlewares/authenticated")
4
5 const api = express.Router()
6
7 api.post("/menu", [md_auth.ensureAuth], MenuController.createMenu)
8
9
10 module.exports = api
```

Ahora vamos a probarla utilizando nuestro insomnia de la siguiente manera, configuramos una nueva carpeta con un nuevo endpoint y hacemos la prueba para validar sin cabecera:



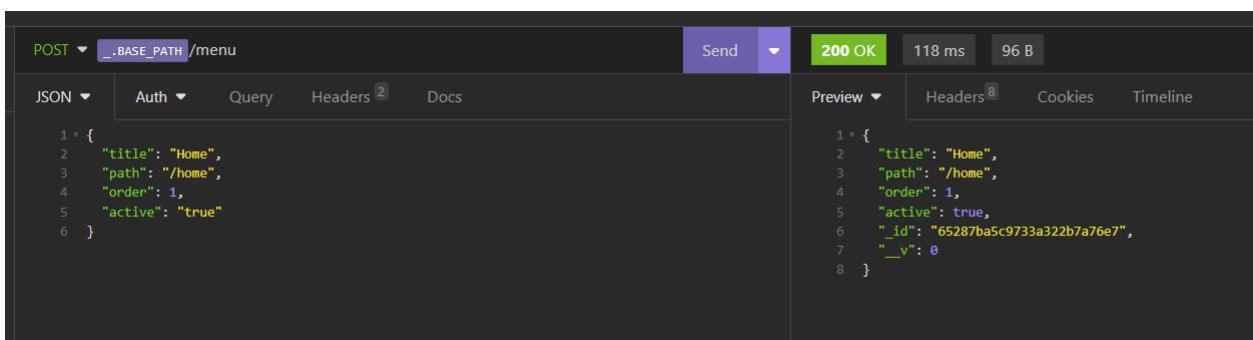
The screenshot shows the insomnia REST client interface. On the left, there's a sidebar with a tree view containing 'localhost', 'Add Cookies', 'Filter', 'Menu' (which has a 'POST createMenu' entry), 'User', and 'Auth'. The main area shows a request configuration for a 'POST' method to '_BASE_PATH/menu'. The 'Headers' tab is selected, showing 'User-Agent: insomnia/2023.5.8'. The 'Send' button is highlighted in purple. To the right, the status bar shows '403 Forbidden', '8.3 ms', and '59 B'. Below the status bar, the 'Preview' tab is open, displaying the JSON response: { "msg": "La petición no tiene la cabecera de autenticación" }.

Y ahora configuramos con la cabecera correcta:



This screenshot shows the same insomnia interface after setting the correct header. In the 'Headers' tab, there is a single entry 'Authorization: Bearer _ACCESS_TOKEN'. The rest of the interface is identical to the previous screenshot, showing the 'POST /menu' endpoint and the 403 Forbidden response.

Y por último pasamos los datos a través de JSON y le damos a enviar:



This screenshot shows the final step where JSON data is passed through the 'Body' tab. The JSON payload is: { "title": "Home", "path": "/home", "order": 1, "active": "true" }. The 'Send' button is highlighted in purple. The status bar shows '200 OK', '118 ms', and '96 B'. The 'Preview' tab shows the successful response: { "title": "Home", "path": "/home", "order": 1, "active": true, "_id": "65287ba5c9733a322b7a76e7", "__v": 0 }.

Y ahora vemos que nos esta creando nuestro menú, para ello vamos a revisar en mongodb:



Y aca podemos observar que ya tenemos nuestra collection menus, vamos a probar a enviar uno nuevo:

Y revisamos en mongo:

De esta manera ya tenemos nuestra creación de menus funcional.



OBTENIENDO LOS MENUS

Ahora vamos a crear el endpoint para obtener el menú completo de nuestra aplicación, para ello nos vamos a continuar en **menu.controller.js** y creamos nuestra función **getMenus**:

```
controllers > js menu.controller.js > ↑ getMenus  
14  
15     async function getMenus(req, res) {  
16         const { active } = req.query  
17  
18         let response = null  
19  
20         if (active === undefined){  
21             response = await Menu.find()  
22         } else {  
23             response = await Menu.find({ active })  
24         }  
25  
26         if(!response) {  
27             res.status(400).send({msg: "No se ha encontrado ningun menu"})  
28         } else {  
29             res.status(200).send(response)  
30         }  
31     }  
32  
33     module.exports = {  
34         createMenu,  
35         getMenus,  
36     }  
37  
38 }
```

Así como realizamos en nuestro controlador de usuarios para obtener el menú lo aplicamos de la misma manera y controlamos en caso de que no se traiga nada con mensajes de status.

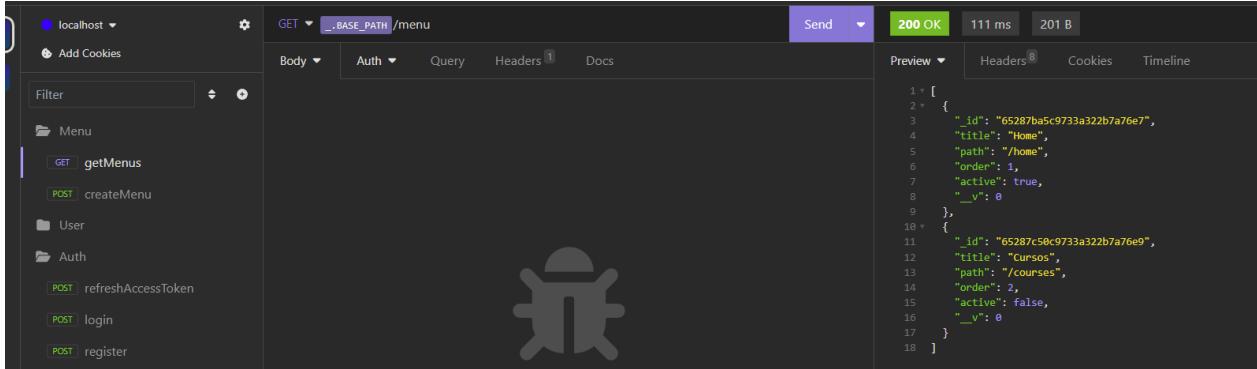
Ahora nos vamos a **menu.router.js** y dejamos nuestra ruta:

```
router > js menu.router.js > ...  
1  const express = require("express")  
2  const MenuController = require("../controllers/menu.controller")  
3  const md_auth = require("../middlewares/authenticated")  
4  
5  const api = express.Router()  
6  
7  api.post("/menu", [md_auth.asureAuth], MenuController.createMenu)  
8  api.get("/menu", MenuController.getMenus)  
9  
10 module.exports = api
```

En nuestra ruta no es necesario que utilizar el middleware de autenticación ya que este menú al obtenerlo lo vamos a mostrar tanto en la web normal y logueados.

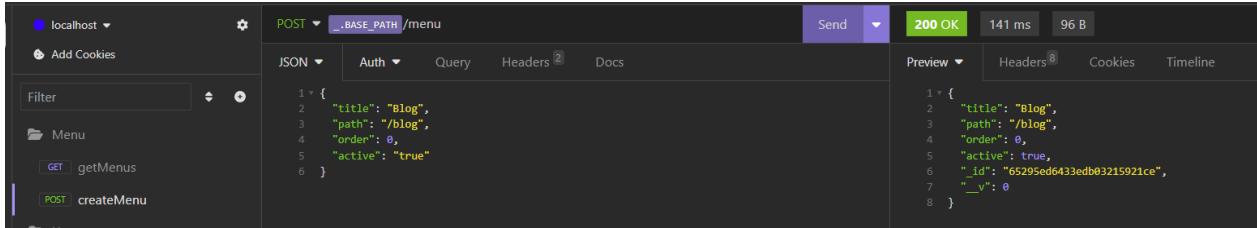


Ahora nos dirigimos a insomnia y allí vamos a probar nuestra nueva ruta:



```
1: [
2:   {
3:     "_id": "65287ba5c9733a322b7a76e7",
4:     "title": "Home",
5:     "path": "/home",
6:     "order": 1,
7:     "active": true,
8:     "__v": 0
9:   },
10:  {
11:    "_id": "65287c50c9733a322b7a76e9",
12:    "title": "Cursos",
13:    "path": "/courses",
14:    "order": 2,
15:    "active": false,
16:    "__v": 0
17:  }
18: ]
```

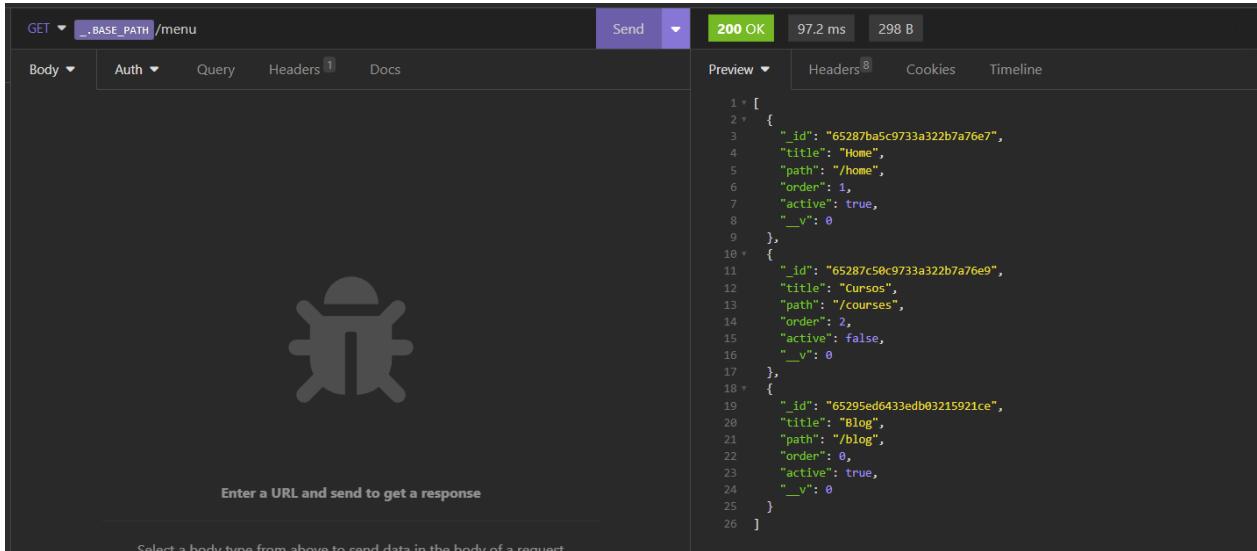
Acá vemos la respuesta de nuestra solicitud y todo va correcto, ahora vamos a probar agregando un nuevo menú:



```
1: {
2:   "title": "Blog",
3:   "path": "/blog",
4:   "order": 0,
5:   "active": true
6: }
```

```
1: {
2:   "title": "Blog",
3:   "path": "/blog",
4:   "order": 0,
5:   "active": true,
6:   "_id": "65295ed6433edb03215921ce",
7:   "__v": 0
8: }
```

Notemos que el menú lo hemos dejado en el orden 0, ahora vamos nuevamente a `getMenus`:



```
1: [
2:   {
3:     "_id": "65287ba5c9733a322b7a76e7",
4:     "title": "Home",
5:     "path": "/home",
6:     "order": 1,
7:     "active": true,
8:     "__v": 0
9:   },
10:  {
11:    "_id": "65287c50c9733a322b7a76e9",
12:    "title": "Cursos",
13:    "path": "/courses",
14:    "order": 2,
15:    "active": false,
16:    "__v": 0
17:  },
18:  {
19:    "_id": "65295ed6433edb03215921ce",
20:    "title": "Blog",
21:    "path": "/blog",
22:    "order": 0,
23:    "active": true,
24:    "__v": 0
25:  }
26: ]
```

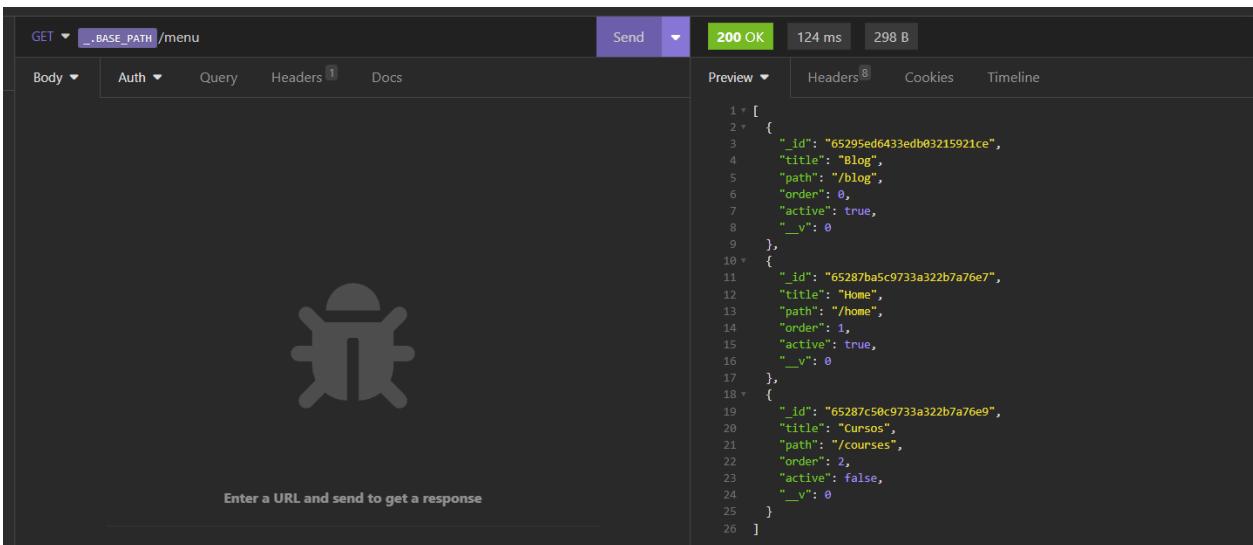
Y acá observamos que al obtenerlo el orden de posición 0 es el tercero, vamos a realizar un ajuste al momento de obtenerlo para que se nos muestre en el orden correcto así que vamos a menu.controller.js y modificamos la respuesta de la siguiente manera:



```
controllers > JS menu.controller.js > getMenus
19
20     if (active === undefined){
21         response = await Menu.find().sort({ order: "asc"})
22     } else {
23         response = await Menu.find({ active }).sort({ order: "asc"})
24     }
25
```

El método `.sort()` se utiliza en una consulta de base de datos para ordenar los resultados según un campo específico. En el contexto de la línea de código que proporcionaste (`Menu.find().sort({ order: "asc" })`), esto significa que los documentos encontrados en la colección **Menu** se ordenarán en función del valor del campo "order" en orden ascendente ("asc").

Ahora volvemos a insomnia a realizar la solicitud y de esta manera obtendremos el orden que necesitábamos:



The screenshot shows the Insomnia REST client interface. The top bar indicates a **GET** request to `.../menu`, with a **200 OK** status code, **124 ms** latency, and **298 B** response size. Below the header, there are tabs for **Body**, **Auth**, **Query**, **Headers**, and **Docs**. The **Body** tab displays the JSON response:

```
1 * [
2 *   {
3 *     "_id": "65295ed6433edb03215921ce",
4 *     "title": "Blog",
5 *     "path": "/blog",
6 *     "order": 0,
7 *     "active": true,
8 *     "__v": 0
9 *   },
10 *   {
11 *     "_id": "65287ba5c9733a322b7a76e7",
12 *     "title": "Home",
13 *     "path": "/home",
14 *     "order": 1,
15 *     "active": true,
16 *     "__v": 0
17 *   },
18 *   {
19 *     "_id": "65287c50c9733a322b7a76e9",
20 *     "title": "Cursos",
21 *     "path": "/courses",
22 *     "order": 2,
23 *     "active": false,
24 *     "__v": 0
25 *   }
26 ]
```



ACTUALIZANDO EL MENU

Vamos a proceder a actualizar el menú mediante su id, para ello vamos a crear dentro de **menu.controller.js** la función **updateMenu**:

```
controllers > js menu.controller.js > <unknown>
33
34     async function updateMenu(req, res) {
35         const { id } = req.params
36         const menuData = req.body
37
38         Menu.findByIdAndUpdate({ _id: id }, menuData, (error) => {
39             if (error) {
40                 res.status(400).send({msg: "Error al actualizar el menu"})
41             } else {
42                 res.status(200).send({msg: "Actualizacion correcta"})
43             }
44         })
45     }
46
47     module.exports = [
48         createMenu,
49         getMenus,
50         updateMenu,
51     ]
```

En esta función observamos que es muy similar a nuestra función de actualizar usuario, recordemos que ubicamos por el id el usuario que queremos actualizar, una vez validados los datos ejecutamos la acción y se procede a cargar, si hay error nos dará el mensaje de status de igual manera si se ejecuta correctamente. Y por último lo exportamos, ahora nos dirigimos a **menu.router.js** y creamos la ruta para nuestro **update**:

```
router > js menu.router.js > ...
1  const express = require("express")
2  const MenuController = require("../controllers/menu.controller")
3  const md_auth = require("../middlewares/authenticated")
4
5  const api = express.Router()
6
7  api.post("/menu", [md_auth.asureAuth], MenuController.createMenu)
8  api.get("/menu", MenuController.getMenus)
9  api.patch("/menu/:id", [md_auth.asureAuth], MenuController.updateMenu)
10
11
12  module.exports = api
```

De esta manera cargamos la ruta utilizando patch, cargamos el middleware de autenticación y ejecutamos la función.

Ahora nos dirigimos a insomnia y creamos para probarlo completamente vamos a generar un nuevo menú primero:



POST `__BASE_PATH__/menu`

JSON **Auth** **Query** **Headers** **Docs**

```
1 v {
2   "title": "Contacto",
3   "path": "/contact",
4   "order": 10,
5   "active": "true"
6 }
```

Send **200 OK** **134 ms** **104 B**

Preview **Headers** **Cookies** **Timeline**

```
1 v {
2   "title": "Contacto",
3   "path": "/contact",
4   "order": 10,
5   "active": true,
6   "_id": "652968e8525ccdbfe9aa4166",
7   "__v": 0
8 }
```

Vamos a tomar el id que nos genero y nos vamos a crear una nueva ruta que se llamará **updateMenus**:

localhost **Add Cookies**

PATCH `__BASE_PATH__/menu/652968e8525ccdbfe9aa4166`

JSON **Auth** **Query** **Headers** **Docs**

1 | ..

Send **403 Forbidden** **1.88 ms** **59 B**

Preview **Headers** **Cookies** **Timeline**

```
1 v {
2   "msg": "La peticion no tiene la cabecera de autenticacion"
3 }
```

Recordemos que para procesar la actualización de los menus debemos estar autenticados para ello configuramos nuestro header:

PATCH `__BASE_PATH__/menu/652968e8525ccdbfe9aa4166`

JSON **Auth** **Query** **Headers** **Docs**

Add Delete All Toggle Description

Content-Type **application/json**

Authorization **Bearer .ACCESS_TOKEN**

Send

Y ahora procedemos a enviar los datos mediante JSON, recordemos que podemos enviar uno o varios al tiempo:

PATCH `__BASE_PATH__/menu/652968e8525ccdbfe9aa4166`

JSON **Auth** **Query** **Headers** **Docs**

```
1 v {
2   "title": "Contacto",
3   "path": "/contact",
4   "order": 5,
5   "active": "true"
6 }
```

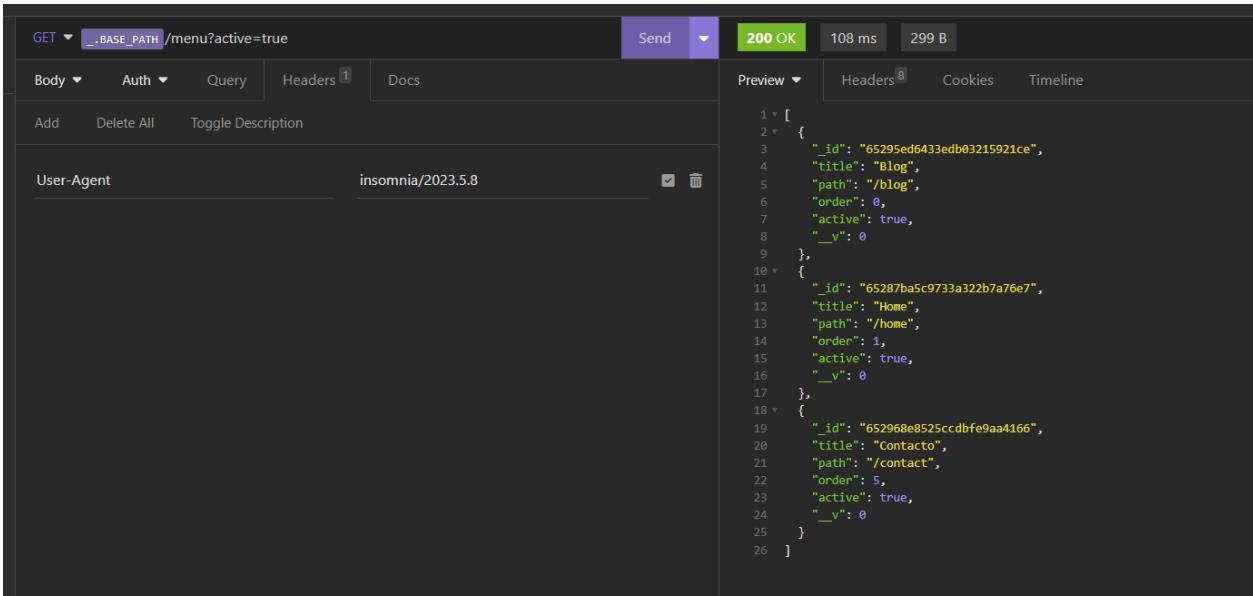
Send **200 OK** **98.4 ms** **32 B**

Preview **Headers** **Cookies** **Timeline**

```
1 v {
2   "msg": "Actualizacion correcta"
3 }
```



Y vemos como se ha actualizado el order, revisamos en getMenus para constatar que si se ha cambiado:



The screenshot shows the Insomnia REST client interface. A GET request is made to `__BASE_PATH__ /menu?active=true`. The response is **200 OK** with a duration of **108 ms** and a size of **299 B**. The **Preview** tab displays the JSON response:

```
1 [  
2 {  
3   "_id": "65295ed6433edb03215921ce",  
4   "title": "Blog",  
5   "path": "/blog",  
6   "order": 0,  
7   "active": true,  
8   "__v": 0  
9 },  
10 {  
11   "_id": "65287ba5c9733a322b7a76e7",  
12   "title": "Home",  
13   "path": "/home",  
14   "order": 1,  
15   "active": true,  
16   "__v": 0  
17 },  
18 {  
19   "_id": "652968e8525ccdbfe9aa4166",  
20   "title": "Contacto",  
21   "path": "/contact",  
22   "order": 5,  
23   "active": true,  
24   "__v": 0  
25 }  
26 ]
```

De esta manera ya tenemos nuestra función de updateMenu lista para poder realizar cualquier edición.

ELIMINACION DE MENU

Vamos a crear nuestra función **deleteMenu** dentro de **menu.controller.js**:

```
controllers > js menu.controller.js > ✖ deleteMenu  
46  
47   async function deleteMenu(req, res) {  
48  
49     const { id } = req.params  
50  
51     Menu.findByIdAndDelete(id, (error) => {  
52       if(error){  
53         res.status(400).send({msg: "Error al eliminar el menu"})  
54       } else {  
55         res.status(200).send({msg: "Menu Eliminado Correctamente"})  
56       }  
57     )  
58   }  
59  
60   module.exports = {  
61     createMenu,  
62     getMenus,  
63     updateMenu,  
64     deleteMenu,  
65   }
```

La estructura que observamos es muy similar a la usada al eliminar nuestro usuario, de esta manera lo que sigue es dirigirnos a crear la ruta en **menu.router.js**:



```

router > js menu.router.js > ...
1 const express = require("express")
2 const MenuController = require("../controllers/menu.controller")
3 const md_auth = require("../middlewares/authenticated")
4
5 const api = express.Router()
6
7 api.post("/menu", [md_auth.asureAuth], MenuController.createMenu)
8 api.get("/menu", MenuController.getMenus)
9 api.patch("/menu/:id", [md_auth.asureAuth], MenuController.updateMenu)
10 api.delete("/menu/:id", [md_auth.asureAuth], MenuController.deleteMenu)
11
12
13 module.exports = api

```

De esta manera configuramos nuestra ruta y ahora procedemos a configurar en insomnia la ruta, primero revisamos que menus tenemos y para mi caso voy a eliminar el menú contacto:

Preview	Headers	Cookies	Timeline
<pre> 1 [2 { 3 "_id": "65295ed6433edb83215921ce", 4 "title": "Blog", 5 "path": "/blog", 6 "order": 0, 7 "active": true, 8 "__v": 0 9 }, 10 { 11 "_id": "65287ba5c9733a322b7a76e7", 12 "title": "Home", 13 "path": "/home", 14 "order": 1, 15 "active": true, 16 "__v": 0 17 }, 18 { 19 "_id": "652960e8525ccdbfe9aa4166", 20 "title": "Contacto", 21 "path": "/contact", 22 "order": 5, 23 "active": true, 24 "__v": 0 25 } 26] </pre>			

Vamos a crear la ruta de la siguiente manera:

Preview	Headers	Cookies	Timeline
<pre> 1 { 2 "msg": "La peticion no tiene la cabecera de autenticacion" 3 } </pre>			

Recordemos que al intentar enviarlo debemos configurar nuestra cabecera:



Todo va bien, ahora revisamos si nuestro menú contacto fue eliminado:

Y solo nos aparece los menus de blog y home, así que ya tenemos funcionando nuestra función de eliminación.

MODELO CURSOS

Vamos a realizar nuestro modelo para la sección de cursos en los cuales vamos a crear de la siguiente manera, primero en **models** vamos a crear **course.model.js**:

```
models > js course.model.js > ...
1  const mongoose = require("mongoose")
2
3  const CourseSchema = mongoose.Schema({
4    title: String,
5    miniature: String,
6    description: String,
7    url: String,
8    price: Number,
9    score: Number
10 })
11
12 module.exports = mongoose.model("Course", CourseSchema)
```

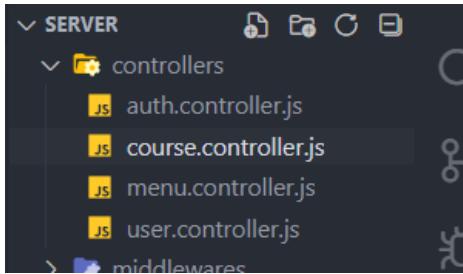
Establecemos nuestro esquema para la db de curso



@hdtoledo

ESTRUCTURA API CURSO

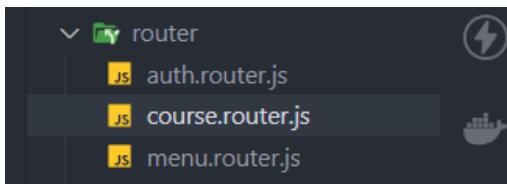
Vamos a crear la estructura de la API del curso para ello vamos a crear el controlador **course.controller.js**



Dentro vamos a dejar lo siguiente:

```
controllers > course.controller.js > ...
1 const Course = require("../models/course.model")
2
3 //Funciones
4
5 |
6 module.exports = {
7
8 }
```

Por el momento vamos a armar la estructura, y ahora creamos la ruta **course.router.js**:



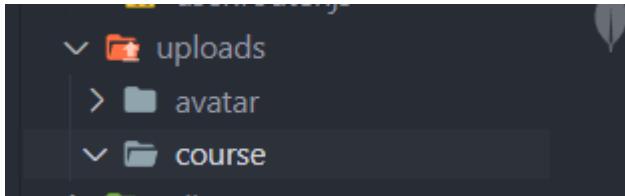
Y dentro dejamos la siguiente estructura:

```
router > course.router.js > <unknown>
1 const express = require("express")
2 const multiparty = require("connect-multiparty")
3 const CourseController = require("../controllers/course.controller")
4 const md_auth = require("../middlewares/authenticated")
5 const md_upload = multiparty({ uploadDir: "/uploads/course" })
6
7
8 const api = express.Router()
9
10 //APIs ...
11
12
13 module.exports = api
```



@hdtoledo

Las **const** que vemos son prácticamente las mismas que hemos implementado en **user.router**, ahora vamos a crear la carpeta **course** dentro de **uploads**:



Ahora nos dirigimos a **app.js** y hacemos la integración:

```
js app.js > courseRoutes
9  // Importar rutas
10 const authRoutes = require("./router/auth.router")
11 const userRoutes = require("./router/user.router")
12 const menuRoutes = require("./router/menu.router")
13 const courseRoutes = require("./router/course.router")
14
```

Importamos nuestra nueva ruta y hacemos la configuración:

```
js app.js > ...
25 // Configurar Rutas
26 app.use(`/api/${API_VERSION}`, authRoutes)
27 app.use(`/api/${API_VERSION}`, userRoutes)
28 app.use(`/api/${API_VERSION}`, menuRoutes)
29 app.use(`/api/${API_VERSION}`, courseRoutes)
30
31
32 module.exports = app
```

Para validar que todo este correcto levantamos el servidor y no debe arrojarnos ningún error:

```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
|
```

Así que con esto ya tenemos la estructura base de cursos.

CREACION DEL CURSO

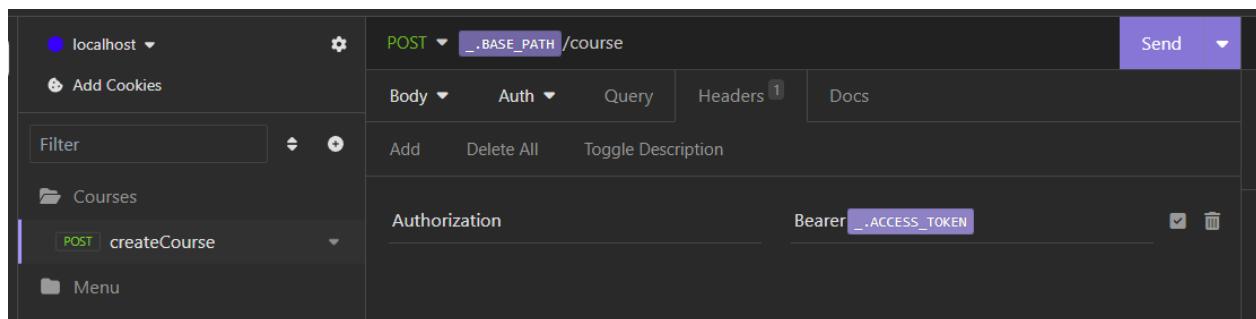
Vamos a empezar por nuestro **course.controller.js** así que lo vamos a estructurar de la siguiente manera:

```
controllers > js course.controller.js > ...
1  const Course = require("../models/course.model")
2  const image = require("../utils/image")
3
4
5  async function createCourse(req, res) {
6      const course = new Course(req.body)
7
8      const imagePath = image.getFilePath(req.files.miniature)
9      course.miniature = imagePath
10
11     course.save((error, courseStored) => {
12         if(error) {
13             res.status(400).send({msg: "Error al crear el curso"})
14         } else {
15             res.status(201).send(courseStored)
16         }
17     })
18 }
19
20 }
21 |
22 module.exports = {
23     createCourse,
24 }
```

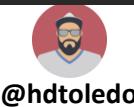
Ahora vamos a **course.router.js** y vamos a dejar la siguiente estructura:

```
router > js course.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const CourseController = require("../controllers/course.controller")
4  const md_auth = require("../middlewares/authenticated")
5  const md_upload = multiparty({ uploadDir: "./uploads/course" })
6
7
8  const api = express.Router()
9
10 api.post("/course", [md_auth.ensureAuth, md_upload], CourseController.createCourse)
11
12
13 module.exports = api
```

De esta manera con esta creación de la ruta ya deberíamos poder crear un curso, para ello utilizamos insomnia y vamos a crear una nueva carpeta con ruta:



The screenshot shows the Insomnia REST Client interface. On the left, there's a sidebar with a tree view. Under the 'Courses' folder, the 'createCourse' endpoint is selected and highlighted in green. At the top, there's a header bar with 'localhost' and a dropdown for 'Send'. Below the header, there's a form for a 'POST' request to '/course'. The 'Authorization' field contains 'Bearer _ACCESS_TOKEN'. There are tabs for 'Body', 'Auth', 'Query', 'Headers', and 'Docs'. At the bottom of the main area, there are buttons for 'Add', 'Delete All', and 'Toggle Description'.



Una vez configurado vamos a pasar a través de **multipart** nuestros datos del curso:

The screenshot shows the Postman interface. On the left, there's a sidebar with 'localhost' selected and a tree view of 'Courses', 'Menu', 'User', and 'Auth'. The main area has a 'POST /course' request. The 'Multipart' tab is selected. The form fields are: title ('Curso de NodeJS'), miniature ('node_curso.png'), description ('Curso basico de NodeJS para aprender a crear tus propias aplicaciones.'), url ('https://codigofacilito.com/cursos/nodejs'), price ('500'), and score ('5'). The 'Send' button is clicked, and the response is shown on the right. It's a '201 Created' response with a timestamp of '137 ms' and a size of '276 B'. The 'Preview' tab shows the JSON response:

```
1: {
2:   "title": "Curso de NodeJS",
3:   "descripcion": "Curso basico de NodeJS para aprender a crear tus propias aplicaciones.",
4:   "url": "https://codigofacilito.com/cursos/nodejs",
5:   "price": 500,
6:   "score": 5,
7:   "_id": "6529a99fb7dc3b7470a79af9",
8:   "miniature": "course/gzYOXWAgPkhekoIrkqFm9DtM.png",
9:   "__v": 0
10: }
```

Ahora para verificar vamos a mongodb y nos encontramos con nuestro nuevo modelo:

The screenshot shows the MongoDB Compass interface. On the left, it shows 'test' as the database and 'courses' as the collection. The 'Find' tab is selected. A query is entered: 'Type a query: { field: 'value' }'. The results show one document: '_id: ObjectId('6529a99fb7dc3b7470a79af9')', 'title: 'Curso de NodeJS'', 'description: 'Curso basico de NodeJS para aprender a crear tus propias aplicaciones.'', 'url: 'https://codigofacilito.com/cursos/nodejs'', 'price: 500', 'score: 5', 'miniature: 'course/gzYOXWAgPkhekoIrkqFm9DtM.png'', and '__v: 0'. The total documents count is 1.

Vamos a comprobar nuevamente con otros datos de curso y validamos:

The screenshot shows the Postman interface again. The 'POST /course' request is shown with the 'Multipart' tab selected. The form fields are: title ('Aprende Node.js y Express - Curso desde cero'), miniature ('node.curso.png'), description ('¡Hola! Si quieres aprender Node.js y Express, estás en el lugar correcto.'), url ('https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/'), price ('300'), and score ('5'). The 'Send' button is clicked, and the response is shown on the right. It's a '201 Created' response with a timestamp of '124 ms' and a size of '355 B'. The 'Preview' tab shows the JSON response:

```
1: {
2:   "title": "Aprende Node.js y Express - Curso desde cero",
3:   "description": "¡Hola! Si quieres aprender Node.js y Express, estás en el lugar correcto.",
4:   "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
5:   "price": 300,
6:   "score": 5,
7:   "_id": "6529aac2b7dc3b7470a79afb",
8:   "miniature": "course/0M2Xljf2P0sLv8t0DDTVj3x.png",
9:   "__v": 0
10: }
```



Validamos en mongodb:

The screenshot shows the MongoDB Compass interface. On the left, there's a sidebar with a '+ Create Database' button, a search bar for namespaces, and a tree view showing the 'test' database with 'courses' and 'menus' collections. The main area is titled 'test.courses' and shows storage details: 36KB, logical data size: 617B, total documents: 2, and index total size: 36KB. Below this are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A 'Filter' input field is present, and a 'Results' button is on the right. The results section is titled 'QUERY RESULTS: 1-2 OF 2' and lists two documents:

```
_id: ObjectId('6529aac2b7dc3b747ea79af9')
title: "Curso de Node.js"
description: "Curso basico de NodeJS para aprender a crear tus propias aplicaciones."
url: "https://codigofacilito.com/cursos/nodejs"
price: 500
score: 5
miniature: "course/gzYXWAgPkhokelrlqfMSDth.png"
__v: 0

_id: ObjectId('6529aac2b7dc3b747ea79af9')
title: "Aprende Node.js y Express - Curso desde cero"
description: "¡Hola! Si quieras aprender Node.js y Express, estás en el lugar correcto."
url: "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-cu-"
price: 300
score: 5
miniature: "course/oWzXljf2P0s1v8tW0DdTvj3x.png"
__v: 0
```

Y de esta manera ya tenemos nuestra función de crear cursos.

OBTENER CURSOS

Para obtener nuestros cursos nos ubicamos en `course.controller.js` y hacemos la función `getCourse`, aca debemos tener en cuenta que debemos paginar la cantidad de datos que vamos a mostrar y para ello vamos a implementar algo luego:

```
controllers > js course.controller.js > unknown
22  async function getCourse(req, res) {
23    Course.find((error, courses) => {
24      if (error) {
25        res.status(400).send({msg: "Error al obtener los cursos"})
26      } else {
27        res.status(200).send(courses)
28      }
29    })
30  }
31
32
33  module.exports = {
34    createCourse,
35    getCourse,
36  }
```

Y ahora nos dirigimos a `course.router.js` y creamos la ruta:



```

router > js course.router.js > ...
1 const express = require("express")
2 const multiparty = require("connect-multiparty")
3 const CourseController = require("../controllers/course.controller")
4 const md_auth = require("../middlewares/authenticated")
5 const md_upload = multiparty({ uploadDir: "./uploads/course" })

6
7
8 const api = express.Router()
9
10 api.post("/course", [md_auth.asureAuth, md_upload], CourseController.createCourse)
11 api.get("/course", CourseController.getCourse)
12
13
14 module.exports = api

```

Recordemos que esta ruta es pública y que tanto los visitantes del sitio como los logueados van a poder verlo, por ello no aplicamos el middleware.

Vamos ahora a insomnia y creamos una nueva petición:

The screenshot shows the Insomnia REST Client interface. On the left, there's a sidebar with a navigation tree: Courses (GET getCourses, POST createCourse), Menu, User, and Auth. The main area has a header with 'localhost' and 'GET _BASE_PATH /course'. Below it are tabs for Body, Auth, Query, Headers, and Docs. A large central area displays a placeholder icon and the text 'Enter a URL and send to get a response'. To the right, the response details are shown: a green '200 OK' button, '116 ms' latency, and '634 B' size. Below this are tabs for Preview, Headers, Cookies, and Timeline. The Preview tab shows the following JSON response:

```

[{"_id": "6529a9fb7dc3b7470a79af9", "title": "Curso de NodeJS", "description": "Curso basico de NodeJS para aprender a crear tus propias aplicaciones.", "url": "https://codigofacilito.com/cursos/nodejs", "price": 500, "score": 5, "miniature": "course/gzYQWAgPhek0lrqFM9Dtt4.png", "__v": 0}, {"_id": "6529aac2b7dc3b7470a79af0", "title": "Aprende Node.js y Express - Curso desde cero", "description": "Hola! Si quieras aprender Node.js y Express, estás en el lugar correcto!", "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/", "price": 300, "score": 5, "miniature": "course/dkzX1jf2PosLv8tMDDdTVj3x.png", "__v": 0}]

```

De esta manera ya obtenemos todos los cursos que acabamos de subir.

PAGINACION DE CURSOS

Ahora imaginemos como paginamos si tuviéramos 100 cursos o más, sería bastante tedioso, así que para ello vamos a utilizar la dependencia de **mongoose paginate**:

The screenshot shows the npm package page for 'mongoose-paginate' at version 5.0.3. The page includes a sidebar with links for Back to search, Information, Website, Repository, Runkit, Tags (latest: 5.0.3), Versions, and Files. The main content area has a title 'mongoose-paginate' with a '5.0.3' badge and a '7 years ago' timestamp. Below this is a note: 'Types are available via @types/mongoose-paginate'. The 'README' section contains the following text:

mongoose-paginate

Pagination plugin for Mongoose

npm v5.0.3 build no longer available

Note: This plugin will only work with Node.js >= 4.0 and Mongoose >= 4.0.

Installation

```
npm install mongoose-paginate
```



En nuestro terminal ejecutamos **yarn add mongoose-paginate**

```
LAPTOPHDMI1 D:\DATA\..\..\server yarn 18.16.0 1ms
{ @hdtolledo } yarn add mongoose-paginate
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
```

Verificamos en nuestro **package.json**:

```
11  },
12  "dependencies": {
13    "bcryptjs": "^2.4.3",
14    "body-parser": "1.20.0",
15    "connect-multiparty": "^2.2.0",
16    "cors": "2.8.5",
17    "express": "4.18.1",
18    "jsonwebtoken": "8.5.1",
19    "mongoose": "6.6.1",
20    "mongoose-paginate": "^5.0.3",
21    "nodemon": "2.0.20"
22  }
23
24 }
```

Ahora levantamos nuestro servidor de nuevo:

```
LAPTOPHDMI1 D:\DATA\..\..\server yarn 18.16.0 1ms
{ @hdtolledo } yarn dev
yarn run v1.22.19
$ nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
|
```

Vamos a ubicarnos en **course.model.js** y vamos a importar nuestra nueva dependencia:

```
models > course.model.js > ...
1 const mongoose = require("mongoose")
2 const mongoosePaginate = require("mongoose-paginate")
3
4 const CourseSchema = mongoose.Schema({
5   title: String,
6   miniature: String,
7   description: String,
8   url: String,
9   price: Number,
10  score: Number
11 })
12
13 CourseSchema.plugin(mongoosePaginate)
14
15 module.exports = mongoose.model("Course", CourseSchema)
```

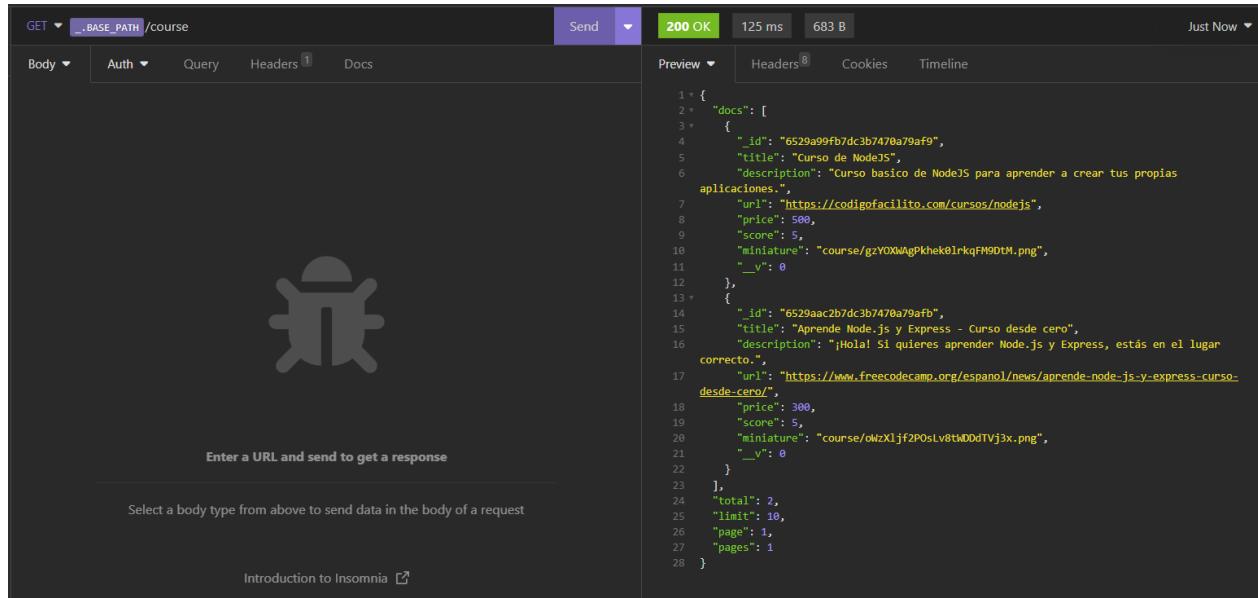


@hdtolledo

y lo iniciamos a través del uso del plugin, de esta manera ya podemos empezar a utilizarlo, para verificarlo nos vamos a **course.controller.js** y dentro de nuestra función **getCourse** hacemos lo siguiente:

```
controllers > js course.controller.js > <unknown>
22  async function getCourse(req, res) {
23
24    const options = {
25      page: 1,
26      limit: 10,
27    }
28
29    Course.paginate({}, options, (error, courses) => {
30      if (error) {
31        res.status(400).send({msg: "Error al obtener los cursos"})
32      } else {
33        res.status(200).send(courses)
34      }
35    })
36
37
38  module.exports = [
39    createCourse,
40    getCourse,
41  ]
42 }
```

De esta manera vamos a ver la implementación de nuestra dependencia, así que ejecutamos de nuevo insomnia y vamos a notar como cambia el contenido que nos llega:



The screenshot shows the Insomnia REST Client interface. A GET request is made to `./BASE_PATH/course`. The response is a 200 OK status with a response time of 125 ms and a body size of 683 B. The response content is a JSON object representing paginated course data:

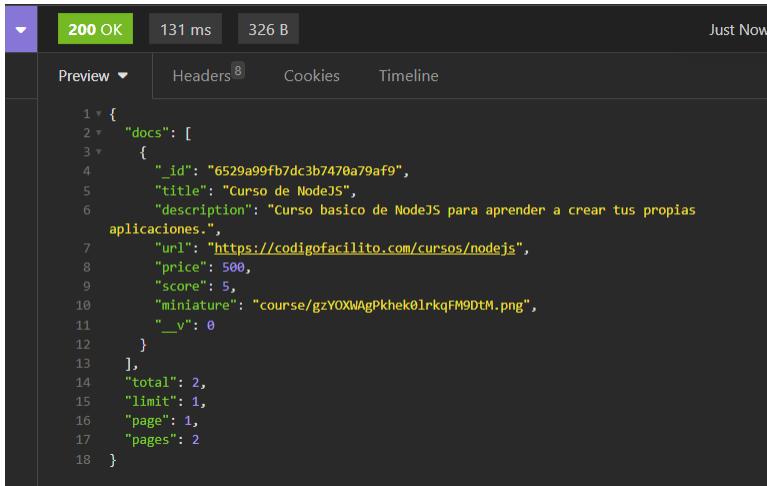
```
{
  "docs": [
    {
      "_id": "6529a09fb7dc3b7470a79af9",
      "title": "Curso de NodeJS",
      "description": "Curso basico de NodeJS para aprender a crear tus propias aplicaciones.",
      "url": "https://codigofacilito.com/cursos/nodejs",
      "price": 500,
      "score": 5,
      "miniature": "course/gzYOXWAgPkhk0lrkqFM9DtM.png",
      "_v": 0
    },
    {
      "_id": "6529aac2b7dc3b7470a79afb",
      "title": "Aprende Node.js y Express - Curso desde cero",
      "description": "¡Hola! Si quieres aprender Node.js y Express, estás en el lugar correcto.",
      "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
      "price": 300,
      "score": 5,
      "miniature": "course/oNzX1jf2P0sLw8tW0DdTVj3x.png",
      "_v": 0
    }
  ],
  "total": 2,
  "limit": 10,
  "page": 1,
  "pages": 1
}
```

Ahora podemos observar cómo nos llega mucho mas organizado los datos y que al finalizar tenemos el total de cursos, el limite de la paginación, la pagina y la cantidad de páginas.

Vamos a modificar un poco el funcionamiento de la siguiente manera:

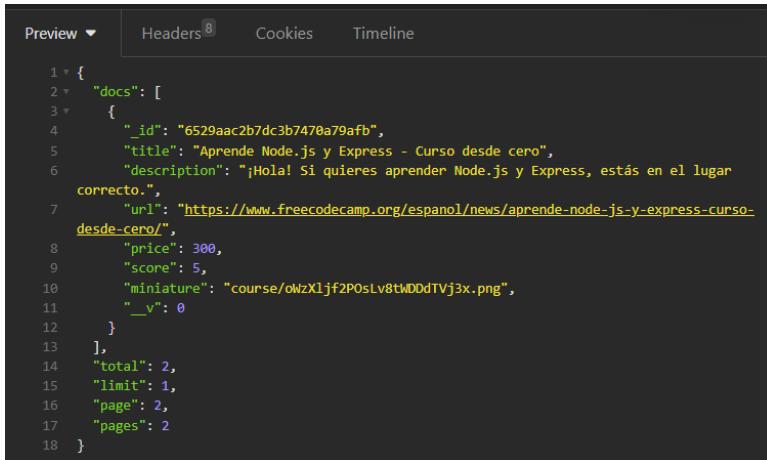
```
22  async function getCourse(req, res) {
23
24    const options = [
25      page: 1,
26      limit: 1,
27    ]
28
29    Course.paginate({}, options, (error, courses) => {
```

En insomnia nos saldría de la siguiente manera:



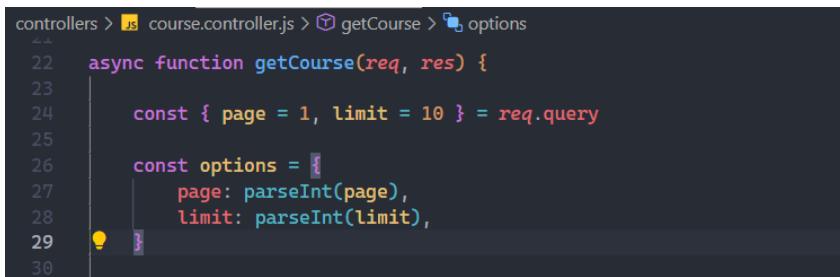
```
1 var {
2   "docs": [
3     {
4       "_id": "6529a99fb7dc3b7470a79af9",
5       "title": "Curso de NodeJS",
6       "description": "Curso básico de NodeJS para aprender a crear tus propias aplicaciones.",
7       "url": "https://codigofacilito.com/cursos/nodejs",
8       "price": 500,
9       "score": 5,
10      "miniature": "course/gzYOXWAgPkhek0lrkqFM9DtM.png",
11      "_v": 0
12    },
13  ],
14  "total": 2,
15  "limit": 1,
16  "page": 1,
17  "pages": 2
18 }
```

En una sola pagina la información, a hora colocamos la siguiente página:



```
1 var {
2   "docs": [
3     {
4       "_id": "6529aac2b7dc3b7470a79afb",
5       "title": "Aprende Node.js y Express - Curso desde cero",
6       "description": "¡Hola! Si quieres aprender Node.js y Express, estás en el lugar correcto.",
7       "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
8       "price": 300,
9       "score": 5,
10      "miniature": "course/oWzXljf2P0sLv8tWDDdTVj3x.png",
11      "_v": 0
12    },
13  ],
14  "total": 2,
15  "limit": 1,
16  "page": 2,
17  "pages": 2
18 }
```

De esta manera podemos modificar nuestra paginación a través de la dependencia, ahora vamos a modificar este proceso ya que debemos tener en cuenta que estos datos de paginación los debe controlar el usuario y los vamos a dejar como variables, vamos a hacer la siguiente modificación:



```
controllers > course.controller.js > getCourse > options
1 const { course } = require('../models')
2 const { Course } = course
3
4 async function getCourse(req, res) {
5   const { title } = req.query
6
7   const courses = await Course.find({ title })
8
9   if (!courses) return res.status(404).json({ message: 'Courses not found' })
10
11   return res.json(courses)
12 }
13
14 module.exports = { getCourse }
```

```
1 var {
2   "docs": [
3     {
4       "_id": "6529aac2b7dc3b7470a79afb",
5       "title": "Aprende Node.js y Express - Curso desde cero",
6       "description": "¡Hola! Si quieres aprender Node.js y Express, estás en el lugar correcto.",
7       "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
8       "price": 300,
9       "score": 5,
10      "miniature": "course/oWzXljf2P0sLv8tWDDdTVj3x.png",
11      "_v": 0
12    },
13  ],
14  "total": 2,
15  "limit": 1,
16  "page": 2,
17  "pages": 2
18 }
```

Establecemos en un objeto nuestras variables y modificamos la const options a través de nuestras nuevas variables. Ahora en nuestro insomnia configuramos de la siguiente manera la ruta:



GET `_.BASE_PATH`/course?page=1&limit=2

Send **200 OK** 119 ms 68 B

Body	Auth	Query	Headers 1	Docs	Preview	Headers 8	Cookies	Timeline
------	------	-------	-----------	------	---------	-----------	---------	----------

```

1 * {
2 >   "docs": [ ↵ 2 ↵ ],
24   "total": 2,
25   "limit": 2,
26   "page": 1,
27   "pages": 1
28 }

```

Nos muestra que tenemos 2 docs y que tenemos un límite de 2 y estoy en la página 1, si nosotros modificamos de nuevo en insomnia:

GET `_.BASE_PATH`/course?page=1&limit=1

Send **200 OK** 98.1 ms 326 B

Body	Auth	Query	Headers 1	Docs	Preview	Headers 8	Cookies	Timeline
------	------	-------	-----------	------	---------	-----------	---------	----------

```

1 * {
2 >   "docs": [ ↵ 1 ↵ ],
14   "total": 2,
15   "limit": 1,
16   "page": 1,
17   "pages": 2
18 }

```

Y si le digo que me voy a la pagina 2:

GET `_.BASE_PATH`/course?page=2&limit=1

Send **200 OK** 98.8 ms 405 B Just Now

Body	Auth	Query	Headers 1	Docs	Preview	Headers 8	Cookies	Timeline
------	------	-------	-----------	------	---------	-----------	---------	----------

```

1 * {
2 >   "docs": [
3 >     {
4       "_id": "6529aac2b7dc3b7470a79afb",
5       "title": "Aprende Node.js y Express - Curso desde cero",
6       "description": "¡Hola! Si quieres aprender Node.js y Express, estás en el lugar correcto.",
7       "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
8       "price": 300,
9       "score": 5,
10      "miniature": "course/oWzX1jf2P0sLv8tMDdTVj3x.png",
11      "__v": 0
12    }
13  ],
14  "total": 2,
15  "limit": 1,
16  "page": 2,
17  "pages": 2
18 }

```

De esta manera ya tenemos implementado nuestro sistema de paginación ya creado con la obtención de cursos.



@hdtoledo

ACTUALIZANDO CURSOS

Vamos a realizar la actualización de nuestros cursos para ello creamos dentro de **course.controller.js** nuestra función **updateCourse**:

```
controllers > course.controller.js > updateCourse
41  async function updateCourse(req, res) {
42      const { id } = req.params
43      const courseData = req.body
44
45      if (req.files.miniature) {
46          const imagePath = image.getFilePath(req.files.miniature)
47          courseData.miniature = imagePath
48      }
49
50      Course.findByIdAndUpdate({ _id: id }, courseData, (error) => {
51          if (error) {
52              res.status(400).send({msg: "Error al actualizar el curso"})
53          } else {
54              res.status(200).send({msg: "Actualizacion correcta"})
55          }
56      })
57  }
58
59  module.exports = {
60      createCourse,
61      getCourse,
62      updateCourse
63 }
```

Como observamos es muy similar a la función utilizada con nuestro usuario, cambiando una pequeña condición en donde preguntamos si la imagen no se actualiza que se mantenga con la misma url, de resto la función actúa de manera muy similar a lo que venimos trabajando, nos dirigimos a nuestro **course.router.js** y cargamos la ruta:

```
router > course.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const CourseController = require("../controllers/course.controller")
4  const md_auth = require("../middlewares/authenticated")
5  const md_upload = multiparty({ uploadDir: "./uploads/course"})
6
7
8  const api = express.Router()
9
10 api.post("/course", [md_auth.asureAuth, md_upload], CourseController.createCourse)
11 api.get("/course", CourseController.getCourse)
12 api.patch("/course/:id", [md_auth.asureAuth, md_upload], CourseController.updateCourse)
13
14
15 module.exports = api
```



@hdtoledo

Para realizar la prueba vamos a crear otro curso en donde este mal su título:

POST .BASE_PATH /course Send 201 Created 148 ms 364 B Just Now

Multipart 6 Auth Query Headers 2 Docs Preview Headers Cookies Timeline

Add Delete All Toggle Description

title	Python para prin
miniature	phantom.png
description	Introducción a Python Aprenda a crear proy
url	https://learn.microsoft.com/es-es/training
price	0
score	5

```
1 var {
2   "title": "Python para prin",
3   "description": "Introducción a Python Aprenda a crear programas y proyectos en Python. Trabaje con cadenas, listas, bucles, diccionarios y funciones.",
4   "url": "https://learn.microsoft.com/es-es/training/patterns/beginner-python/",
5   "price": 0,
6   "score": 5,
7   "_id": "652abc994001b82a70f44eb9",
8   "miniature": "course/IouMRYENLCI-j30gGuRQF16k.png",
9   "__v": 0
10 }
```

Ahora vamos a probarlo en nuestro insomnia de la siguiente manera creando la ruta para updateCourse y colocando el id de nuestro nuevo curso:

The screenshot shows a POSTMAN interface with the following details:

- URL:** PATCH `_base_path /course/652abc994001b82a70f44eb9`
- Status:** 200 OK, 137 ms, 32 B
- Multipart:** Auth, Query, Headers, Docs
- Body (raw):**

```
1 + {  
2     "msg": "Actualización correcta"  
3 }
```
- Request Body Fields:**
 - title:** Python para principiantes - Nivel 0
 - miniature:** node.curso.png
 - description:** ¡Hola! Si quieres aprender Node.js y Expr
 - url:** <https://www.freecodecamp.org/espagnol/>

Ahora vamos a verificar si es correcto:

GET `_BASE_PATH` /course?page=1&limit=10

Send ▾

200 OK | 131 ms | 1067 B | Just Now ▾

Body ▾ Auth ▾ Query Headers 1 Docs

Preview ▾ Headers 8 Cookies Timeline

```
aplicaciones."
7   "url": "https://codigofacilito.com/cursos/nodejs",
8   "price": 500,
9   "score": 5,
10  "miniature": "course/gzYXWAgPkhek0lrkqFM9DtM.png",
11  "_v": 0
12 },
13 +
14 {
15   "_id": "6529aac2b7dc3b7470a79af",
16   "title": "Aprende Node.js y Express - Curso desde cero",
17   "description": "¡Hola! Si quieras aprender Node.js y Express, estás en el lugar correcto.",
18   "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
19   "price": 300,
20   "score": 5,
21   "miniature": "course/oMzXljf2P0sLv8tWDDTVj3x.png",
22   "_v": 0
23 },
24 {
25   "_id": "652abc994001b82a70f44eb9",
26   "title": "Python para principiantes - Nivel 0",
27   "description": "Introducción a Python Aprenda a crear programas y proyectos en Python. Trabaje con cadenas, listas, bucles, diccionarios y funciones.",
28   "url": "https://learn.microsoft.com/es-es/training/paths/beginner-python/",
29   "price": 0,
30   "score": 5,
31   "miniature": "course/IouWRYENLCI-j30gGuRQF16k.png",
32   "_v": 0
33 }
34 
```

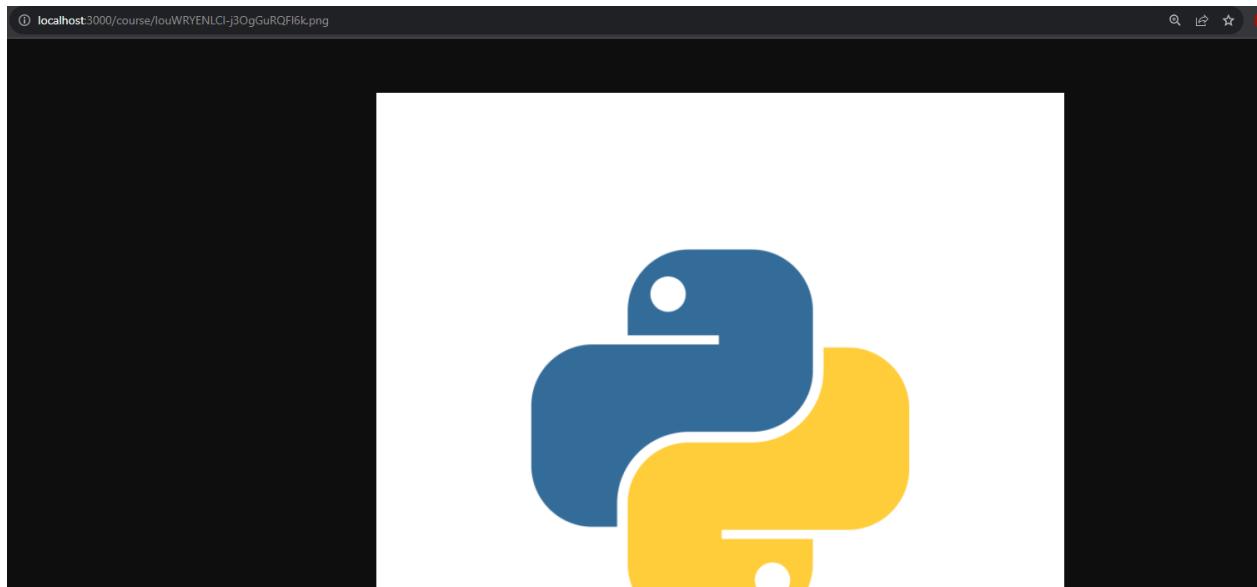


@hdtoledo

Y en mongodb:

```
_id: ObjectId('652abc994001b82a70f44eb9')
title: "Python para principiantes - Nivel 0"
description: "Introducción a Python Aprenda a crear programas y proyectos en Python..."
url: "https://learn.microsoft.com/es-es/training/patterns/beginner-python/"
price: 0
score: 5
miniature: "course/IouWRYENLCI-j30gGuRQFl6k.png"
__v: 0
```

Ahora vamos a comprobar si la url de nuestros cursos funcionan, copiamos la url y la colocamos en nuestro navegador y verificamos:



De esta manera estamos verificando que nuestra función de actualizar ha sido correctamente creada.



ELIMINAR CURSOS

Ahora vamos a seguir con la función de **deleteCourse** dentro de nuestro **course.controller.js**:

```
controllers > [js] course.controller.js > 📁 <unknown>
~~
59  √  async function deleteCourse(req, res) {
60      const { id } = req.params
61
62  √      Course.findByIdAndDelete(id, (error) => {
63      if (error) {
64          res.status(400).send({msg: "Error al eliminar el curso"})
65  √      } else {
66          res.status(200).send({msg: "Curso Eliminado"})
67      }
68  √  })
69  }
70
71  √  module.exports = {
72      createCourse,
73      getCourse,
74      updateCourse,
75      deleteCourse,
76  }

```

Como observamos la función de eliminación es similar a la de usuario solo que acá lo hacemos con el id de cursos, ahora vamos a crear nuestra ruta en **course.router.js**:

```
router > [js] course.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const CourseController = require("../controllers/course.controller")
4  const md_auth = require("../middlewares/authenticated")
5  const md_upload = multiparty({ uploadDir: "./uploads/course"})
6
7
8  const api = express.Router()
9
10 api.post("/course", [md_auth.asureAuth, md_upload], CourseController.createCourse)
11 api.get("/course", CourseController.getCourse)
12 api.patch("/course/:id", [md_auth.asureAuth, md_upload], CourseController.updateCourse)
13 api.delete("/course/:id", [md_auth.asureAuth], CourseController.deleteCourse)
14
15
16  module.exports = api

```

Ahora vamos a comprobar que funciona para ello vamos a crear un curso nuevo en insomnia:



@hdtoledo

```

1: {
2:   "title": "Academia de profesores de Minecraft: Education",
3:   "description": "Aprendizaje de Minecraft Education",
4:   "url": "https://learn.microsoft.com/es-es/training/paths/minecraft-teacher-academy",
5:   "price": 0,
6:   "score": 5,
7:   "_id": "652ac04d79651500110b1010",
8:   "miniature": "course/Si8EYfeQu54MjYMo570FAZk.png",
9:   "__v": 0
10: }

```

Ahora creamos una nueva ruta para nuestro deleteCourse en insomnia:

```

1: {
2:   "msg": "Curso Eliminado"
3: }

```

Verificamos en mongodb e insomnia:

```

1: {
2:   "_id": "652aaac2b7dc3b7478a79af9",
3:   "title": "Curso de NodeJS",
4:   "description": "Curso basico de NodeJS para aprender a crear tus propias aplicaciones.",
5:   "url": "https://codigofacilito.com/cursos/nodejs",
6:   "price": 500,
7:   "score": 5,
8:   "miniature": "course/gzYOWAgPikeh@LrkqFM90tN.png",
9:   "__v": 0
10: },
11: {
12:   "_id": "652aaac2b7dc3b7478a79af9",
13:   "title": "Aprende Node.js y Express - Curso desde cero",
14:   "description": "¡Hola! Si quieras aprender Node.js y Express, estás en el lugar correcto.",
15:   "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
16:   "price": 300,
17:   "score": 5,
18:   "miniature": "course/oMzX1jfzPOsLv8tM000IVj3x.png",
19:   "__v": 0
20: },
21: {
22:   "_id": "652abc9949001b82a0f44eb9",
23:   "title": "Python para principiantes - Nivel 9",
24:   "description": "Introducción a Python Aprenda a crear programas y proyectos en Python. Trabaje con cadenas, listas, bucles, diccionarios y funciones.",
25:   "url": "https://learn.microsoft.com/es-es/training/paths/beginner-python",
26:   "price": 0,
27:   "score": 5,
28:   "miniature": "course/InslHRYENL1-j30gGuQF16k.png",
29:   "__v": 5
30: }

```

```

1: {
2:   "_id": "652aaac2b7dc3b7478a79af9",
3:   "title": "Curso de NodeJS",
4:   "description": "Curso basico de NodeJS para aprender a crear tus propias aplicaciones.",
5:   "url": "https://codigofacilito.com/cursos/nodejs",
6:   "price": 500,
7:   "score": 5,
8:   "miniature": "course/gzYOWAgPikeh@LrkqFM90tN.png",
9:   "__v": 0
10: },
11: {
12:   "_id": "652aaac2b7dc3b7478a79af9",
13:   "title": "Aprende Node.js y Express - Curso desde cero",
14:   "description": "¡Hola! Si quieras aprender Node.js y Express, estás en el lugar correcto.",
15:   "url": "https://www.freecodecamp.org/espanol/news/aprende-node-js-y-express-curso-desde-cero/",
16:   "price": 300,
17:   "score": 5,
18:   "miniature": "course/oMzX1jfzPOsLv8tM000IVj3x.png",
19:   "__v": 0
20: },
21: {
22:   "_id": "652abc9949001b82a0f44eb9",
23:   "title": "Python para principiantes - Nivel 9",
24:   "description": "Introducción a Python Aprenda a crear programas y proyectos en Python. Trabaje con cadenas, listas, bucles, diccionarios y funciones.",
25:   "url": "https://learn.microsoft.com/es-es/training/paths/beginner-python",
26:   "price": 0,
27:   "score": 5,
28:   "miniature": "course/InslHRYENL1-j30gGuQF16k.png",
29:   "__v": 5
30: }

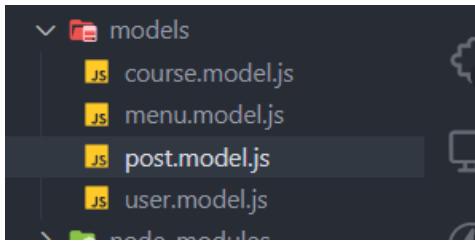
```

De esta manera ya terminamos nuestra función de eliminación.



MODELO POST PARA EL BLOG

Ahora vamos a desarrollar el modelo de datos para nuestro blog de la aplicación, lo primero es crear nuestro modelo **post.model.js** dentro de **models**



Ahora dejamos de la siguiente manera nuestro modelo:

```
models > post.model.js > <unknown>
1  const mongoose = require("mongoose")
2  const mongoosePaginate = require("mongoose-paginate")
3
4  const PostSchema = mongoose.Schema({
5      title: String,
6      miniature: String,
7      content: String,
8      path: {
9          type: String,
10         unique: true,
11     },
12     created_at: Date,
13 })
14
15 PostSchema.plugin(mongoosePaginate)
16
17 module.exports = mongoose.model("Post", PostSchema)
```

Dentro de nuestro modelo dejamos implementado el sistema de paginación ya que tendremos una gran cantidad de post en esta sección. Revisamos en nuestro servidor que todo vaya bien sin errores:

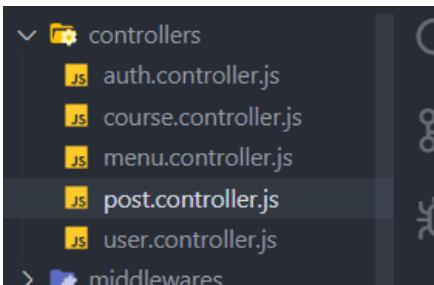
```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
```



@hdtoledo

ESTRUCTURA API BLOG

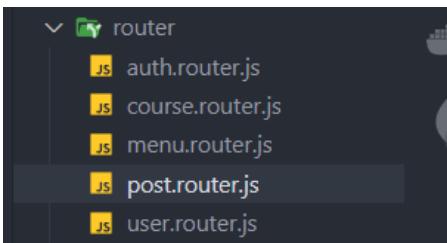
Vamos a crear nuestro controlador **post.controller.js** dentro de **controllers**:



Y dejamos de la siguiente manera nuestra estructura inicial:

```
controllers > post.controller.js > <unknown>
1  const Post = require("../models/post.model")
2
3  //Funciones ...
4
5
6  module.exports = [
7  |
8  ]
```

Ahora creamos nuestra ruta **post.router.js**:



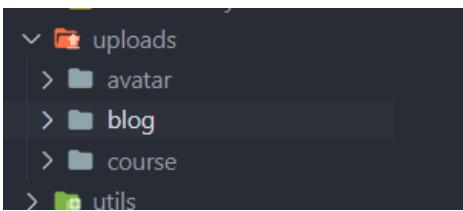
Y lo dejamos con la siguiente estructura:

```
router > post.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const PostController = require("../controllers/post.controller")
4  const md_auth = require("../middlewares/authenticated")
5
6  const md_upload = multiparty({ uploadDir: "./uploads/blog"})
7  const api = express.Router()
8
9  //Rutas ...
10
11
12  module.exports = api
```



@hdtoledo

Ahora creamos nuestra carpeta para **blog** dentro de **uploads**:



Y por último configuramos nuestro **app.js** importamos nuestra ruta:

```
js app.js > postRoutes
 9 // Importar rutas
10 const authRoutes = require("./router/auth.router")
11 const userRoutes = require("./router/user.router")
12 const menuRoutes = require("./router/menu.router")
13 const courseRoutes = require("./router/course.router")
14 const postRoutes = require("./router/post.router")
15
```

Y configuramos la ruta:

```
js app.js > ...
25
26 // Configurar Rutas
27 app.use(`/api/${API_VERSION}`, authRoutes)
28 app.use(`/api/${API_VERSION}`, userRoutes)
29 app.use(`/api/${API_VERSION}`, menuRoutes)
30 app.use(`/api/${API_VERSION}`, courseRoutes)
31 app.use(`/api/${API_VERSION}`, postRoutes)
32
33
34 module.exports = app
```

Revisamos que nuestro servidor este correctamente funcionando:

```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
|
```

Y aquí ya dejamos nuestra configuración inicial.



@hdtoledo

CREACION DE POST

Ahora vamos a continuar con la función de creación del post, nos dirigimos a **post.controller.js** y vamos a crear nuestra función **createPost**:

```
controllers > js post.controller.js > ...
1  const Post = require("../models/post.model")
2  const image = require("../utils/image")
3
4  function createPost(req, res) {
5      const post = new Post(req.body)
6      post.created_at = new Date()
7
8      const imagePath = image.getFilePath(req.files.miniature)
9      post.miniature = imagePath
10
11     post.save((error, postStored) => {
12         if (error) {
13             res.status(400).send({msg: "Error al crear el post"})
14         } else {
15             res.status(201).send(postStored)
16         }
17     })
18 }
19
20 module.exports = {
21     createPost,
22 }
```

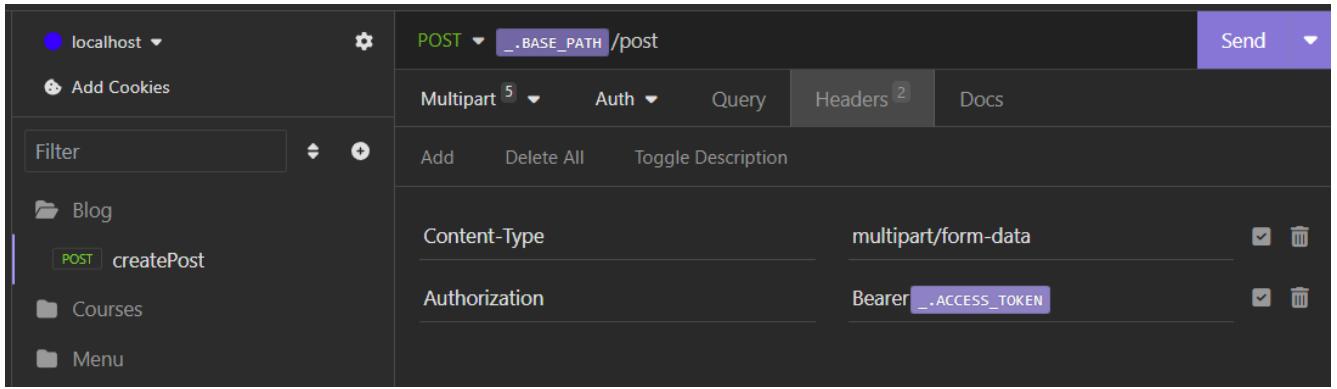
Creamos nuestra función del post y definimos path de imagen y fecha de publicación, ahora vamos a crear nuestra ruta dentro de **post.router.js**:

```
router > js post.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const PostController = require("../controllers/post.controller")
4  const md_auth = require("../middlewares/authenticated")
5
6  const md_upload = multiparty({ uploadDir: "./uploads/blog" })
7  const api = express.Router()
8
9
10 api.post("/post", [md_auth.ensureAuth, md_upload], PostController.createPost)
11
12
13 module.exports = api
```



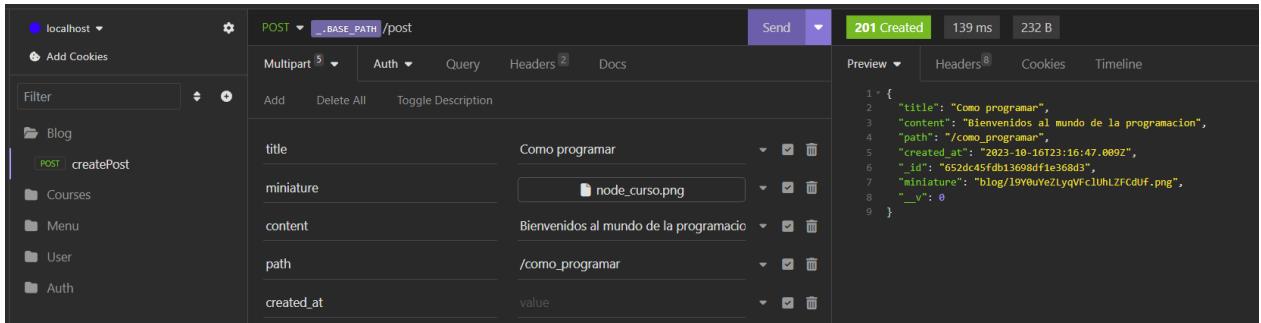
@hdtoledo

Ahora en insomnia vamos a crear una carpeta nueva con su ruta:



The screenshot shows the insomnia REST client interface. On the left, there's a sidebar with a tree view containing 'Blog' (selected), 'Courses', 'Menu', and 'Auth'. The main area shows a POST request to `_BASE_PATH/post`. The 'Headers' tab is active, showing 'Content-Type: multipart/form-data' and 'Authorization: Bearer _ACCESS_TOKEN'. The 'Send' button is at the top right.

Recordemos pasar la configuración de nuestro base path y en los headers aplicamos auth y Access token, ahora aplicamos la configuración de multipart:

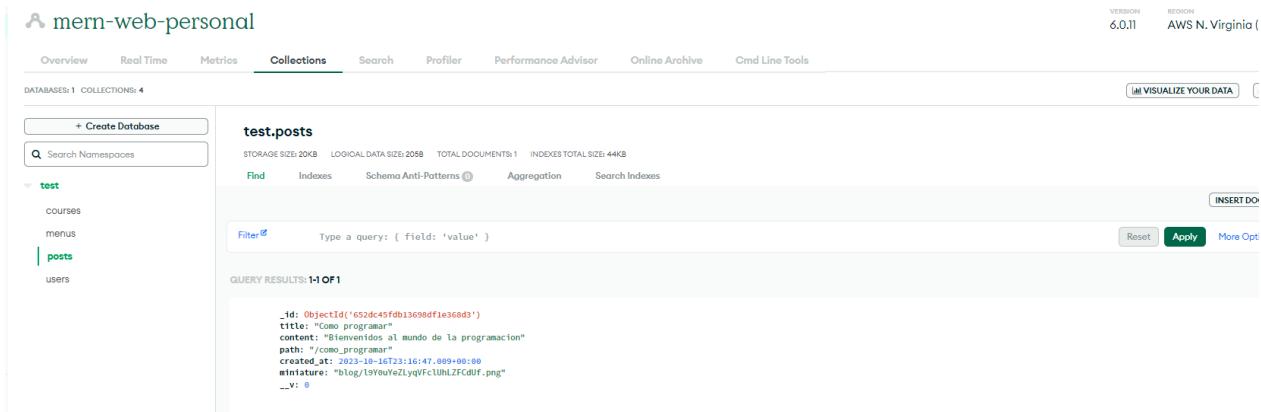


The screenshot shows the insomnia REST client interface after sending the POST request. The status bar indicates a **201 Created** response with 139 ms and 232 B. The 'Preview' tab shows the JSON response body:

```
1 {  
2   "title": "Como programar",  
3   "content": "Bienvenidos al mundo de la programacion",  
4   "path": "/como_programar",  
5   "created_at": "2023-10-16T23:16:47.009Z",  
6   "_id": "652dc45fd13698df1e368d3",  
7   "miniature": "blog/19y0uYeZlyqVFclUhlZFCduF.png",  
8   "__v": 0  
9 }
```

Nota: al colocar la ruta del **path** no debemos colocar el **slash "/"**.

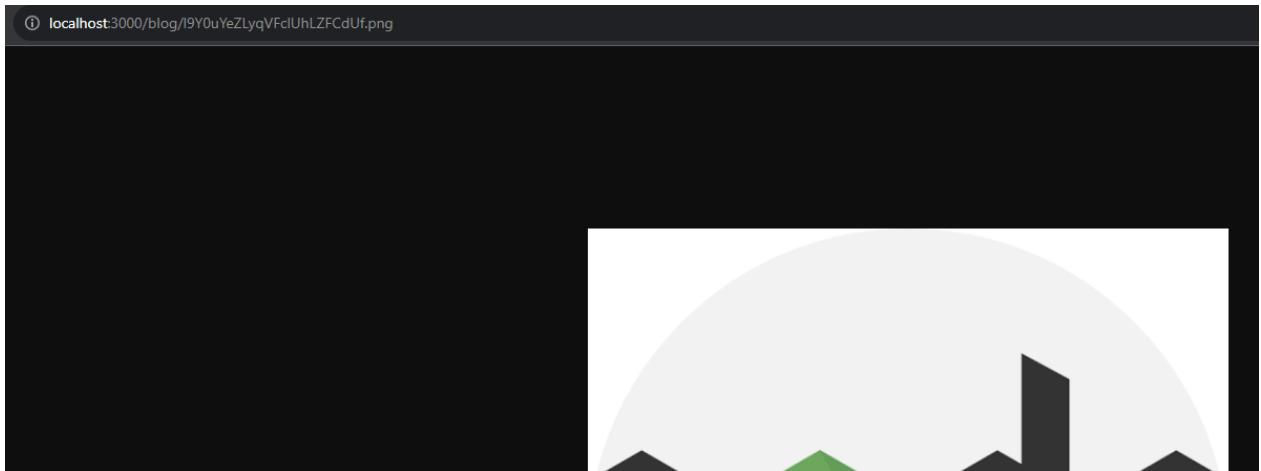
Al enviarlo nos debe devolver nuestro objeto, vamos a verificarlo en mongodb y tambien vamos a verificar copiando y pegando la dirección de la miniatura en nuestro navegador:



The screenshot shows the MongoDB Compass interface. On the left, the database 'mern-web-personal' is selected, with the 'test' database expanded to show 'courses', 'menus', 'posts' (selected), and 'users'. The 'Collections' tab is active, showing the 'posts' collection. The collection details pane shows 'STORAGE SIZE: 20KB', 'LOGICAL DATA SIZE: 2058', 'TOTAL DOCUMENTS: 1', and 'INDEXES TOTAL SIZE: 44KB'. The 'Find' tab is selected, and the query builder shows a single result: '_id: 652dc45fd13698df1e368d3'. The results pane shows the document content:

```
_id: 652dc45fd13698df1e368d3  
title: "Como programar"  
content: "Bienvenidos al mundo de la programacion"  
path: "/como_programar"  
created_at: 2023-10-16T23:16:47.009Z  
miniature: "blog/19y0uYeZlyqVFclUhlZFCduF.png"  
__v: 0
```





Vamos a dejar cargada una publicación más como ejercicio:

POST `$_BASE_PATH/post`

Multipart	Auth	Query	Headers	Docs	Send	201 Created	129 ms	232 B
Add	Delete All	Toggle Description						
title	Como programar nivel 2	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>						
miniature		<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>						
content	El mundo de la programacion 2	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>						
path	/como_programar_2	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>						
created_at	value	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>						

Preview Headers Cookies Timeline

```

1 var {
2   "title": "Como programar nivel 2",
3   "content": "El mundo de la programacion 2",
4   "path": "/como_programar_2",
5   "created_at": "2023-10-16T23:22:32.225Z",
6   "_id": "652dc5b8db13698df1e368d5",
7   "miniature": "blog/l9Y0uYeZLyqVFcUhLZFCdUf.png",
8   "__v": 0
9 }

```

Filter Type a query: { field: 'value' } Reset Apply More

QUERY RESULTS: 1-2 OF 2

```

_id: ObjectId('652dc45fd8db13698df1e368d3')
title: "Como programar"
content: "Bienvenidos al mundo de la programacion"
path: "/como_programar"
created_at: 2023-10-16T23:16:47.009+00:00
miniature: "blog/l9Y0uYeZLyqVFcUhLZFCdUf.png"
__v: 0

_id: ObjectId('652dc5b8db13698df1e368d5')
title: "Como programar nivel 2"
content: "El mundo de la programacion 2"
path: "/como_programar_2"
created_at: 2023-10-16T23:22:32.225+00:00
miniature: "blog/n1ObUyGbjLeHr8xSMlGiwEP.png"
__v: 0

```

De esta manera comprobamos y dejamos lista nuestra función de crear post.



OBTENCION Y PAGINACION DE LOS POST

Vamos a crear nuestra función `getPosts` dentro de `post.controller.js`:

```
controllers > js post.controller.js > ...
20
21     function getPosts(req, res) {
22         const { page = 1, limit = 10 } = req.query
23
24         const options = {
25             page: parseInt(page),
26             limit: parseInt(limit),
27             sort: { created_at: "desc" }
28         }
29
30         Post.paginate({}, options, (error, postsStored) => {
31             if (error) {
32                 res.status(400).send({msg: "Error al obtener los post"})
33             } else {
34                 res.status(200).send(postsStored)
35             }
36         })
37     }
38
39
40     module.exports = {
41         createPost,
42         getPosts,
43     }

```

De esta manera aplicamos de manera similar nuestra paginación dentro de la obtención del post, ahora vamos a crear la ruta dentro de `post.router.js`:

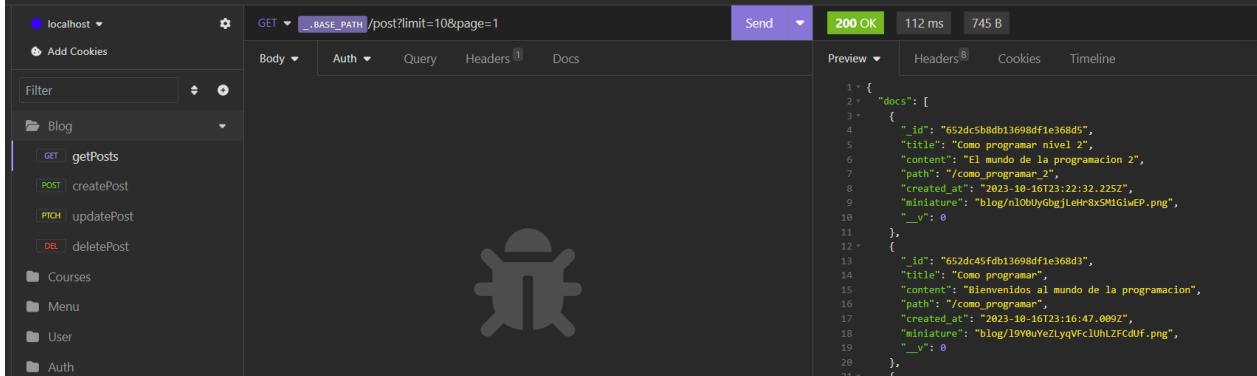
```
router > js post.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const PostController = require("../controllers/post.controller")
4  const md_auth = require("../middlewares/authenticated")
5
6  const md_upload = multiparty({ uploadDir: "./uploads/blog" })
7  const api = express.Router()
8
9
10 api.post("/post", [md_auth.asureAuth, md_upload], PostController.createPost)
11 api.get("/post", PostController.getPosts)
12
13
14 module.exports = api

```



@hdtoledo

Recordemos que es un **get** y debe ser publico así que ahora nos vamos a insomnia y creamos la ruta y probamos:



```
1: {
2:   "docs": [
3:     {
4:       "_id": "652dc5b8db13698df1e368d5",
5:       "title": "Como programar nivel 2",
6:       "content": "El mundo de la programacion 2",
7:       "path": "/como_programar_2",
8:       "created_at": "2023-10-16T23:22:32.225Z",
9:       "miniature": "blog/n10bUygbgjLehr8xSM1GiwEP.png",
10:      "__v": 0
11    },
12    {
13      "_id": "652dc45fd8b13698df1e368d3",
14      "title": "Como programan",
15      "content": "Bienvenidos al mundo de la programación",
16      "path": "/como_programan",
17      "created_at": "2023-10-16T23:16:47.009Z",
18      "miniature": "blog/19Y8uYeZLyqVFclUhLZFcduf.png",
19      "__v": 0
20  },
21  ],
22  "meta": {
23    "total": 2,
24    "limit": 10,
25    "page": 1,
26    "pages": 1
27  }
28}
```

De esta manera ya tenemos nuestra función completa.

ACTUALIZACION DE LOS POST

Vamos a crear nuestra función **updatePost** dentro de **post.controller.js**:

```
controllers > post.controller.js >  <unknown>
39  function updatePost (req, res) {
40    const { id } = req.params
41    const postData = req.body
42
43    if (req.files.miniature) {
44      const imagePath = image.getFilePath(req.files.miniature)
45      postData.miniature = imagePath
46    }
47
48    Post.findByIdAndUpdate({ _id: id }, postData, (error) => {
49      if (error) {
50        res.status(400).send({msg: "Error al actualizar el post"})
51      } else {
52        res.status(200).send({msg: "Actualizacion correcta"})
53      }
54    })
55  }
56
57  module.exports = {
58    createPost,
59    getPosts,
60    updatePost,
61  }
```

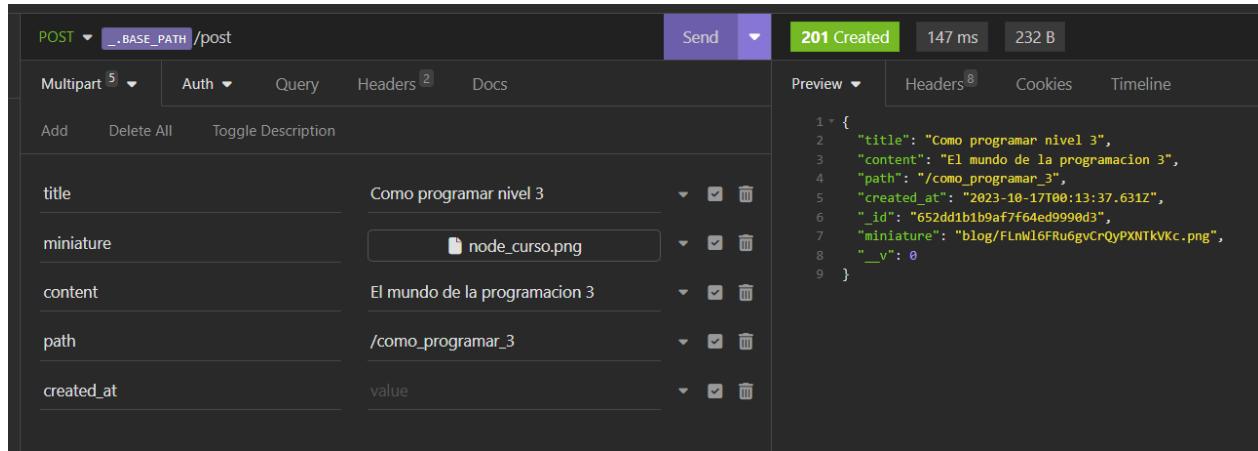


@hdtoledo

Nuestra función es muy similar a la que utilizamos anteriormente, ahora procedemos a crear la ruta dentro de **post.router.js**

```
router > js post.router.js > ...
1  const express = require("express")
2  const multiparty = require("connect-multiparty")
3  const PostController = require("../controllers/post.controller")
4  const md_auth = require("../middlewares/authenticated")
5
6  const md_upload = multiparty({ uploadDir: "./uploads/blog" })
7  const api = express.Router()
8
9
10 api.post("/post", [md_auth.asureAuth, md_upload], PostController.createPost)
11 api.get("/post", PostController.getPosts)
12 api.patch("/post/:id", [md_auth.asureAuth, md_upload], PostController.updatePost)
13
14
15 module.exports = api
```

Ahora vamos a nuestro insomnia y creamos un nuevo post:



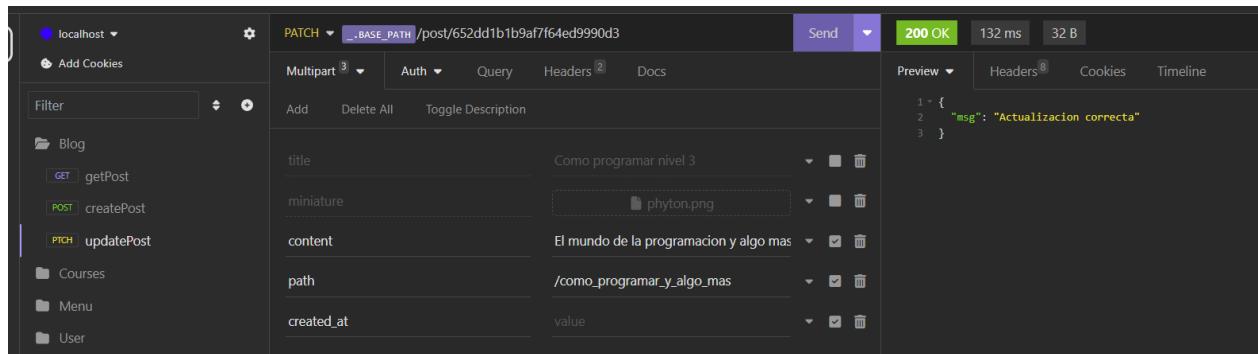
The screenshot shows the insomnia API client interface. A POST request is made to `_.BASE_PATH /post`. The request body contains the following data:

```
title: "Como programar nivel 3"
miniature: "node_curso.png"
content: "El mundo de la programacion 3"
path: "/como_programar_3"
created_at: "value"
```

The response is a **201 Created** status with a response body:

```
1: {
2:   "title": "Como programar nivel 3",
3:   "content": "El mundo de la programacion 3",
4:   "path": "/como_programar_3",
5:   "created_at": "2023-10-17T00:13:37.631Z",
6:   "_id": "652dd1b1b0af7f64ed9990d3",
7:   "miniature": "blog/FLnWl6FRu6gvCrQyPXNTkVKc.png",
8:   "__v": 0
9: }
```

Vamos a utilizar el id de nuestro post en la ruta nueva:



The screenshot shows the insomnia API client interface. A PATCH request is made to `_.BASE_PATH /post/652dd1b1b0af7f64ed9990d3`. The request body contains the following data:

```
title: "Como programar nivel 3"
miniature: "python.png"
content: "El mundo de la programacion y algo mas"
path: "/como_programar_y_algo_mas"
created_at: "value"
```

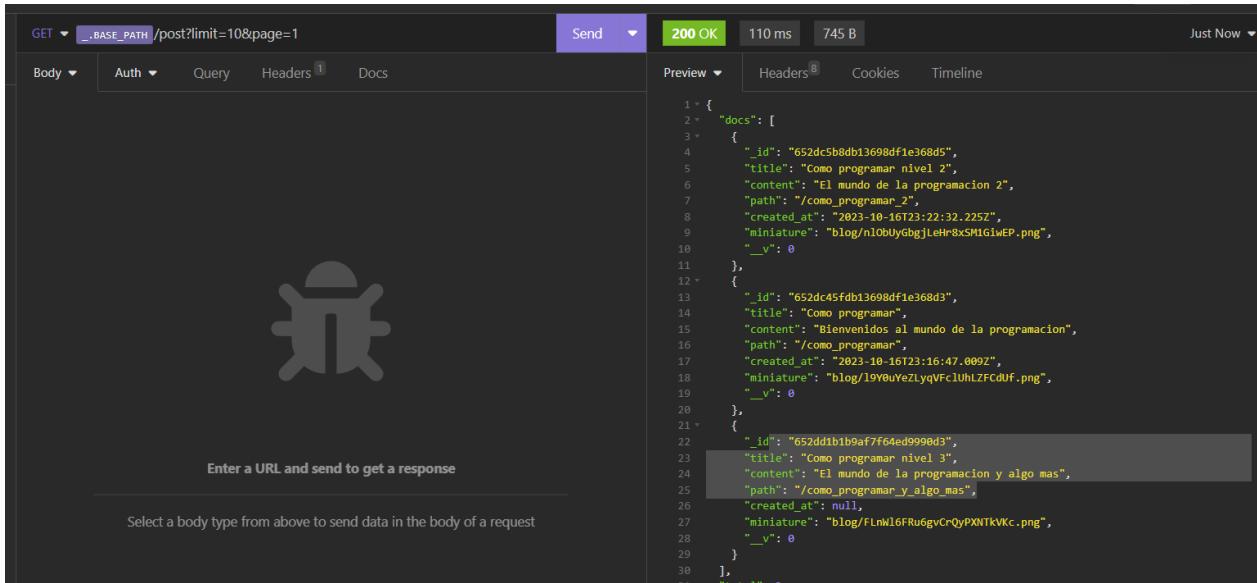
The response is a **200 OK** status with a response body:

```
1: {
2:   "msg": "Actualizacion correcta"
3: }
```



@hdtoledo

Revisamos en nuestro `getPost`:



The screenshot shows a browser-based API testing interface. At the top, it says "GET _BASE_PATH /post?limit=10&page=1". Below that, there are tabs for "Body", "Auth", "Query", "Headers", and "Docs". The "Send" button is purple and has a dropdown arrow. To its right, it says "200 OK", "110 ms", and "745 B". On the far right, it says "Just Now". Under the "Preview" tab, there is a code block showing the JSON response:

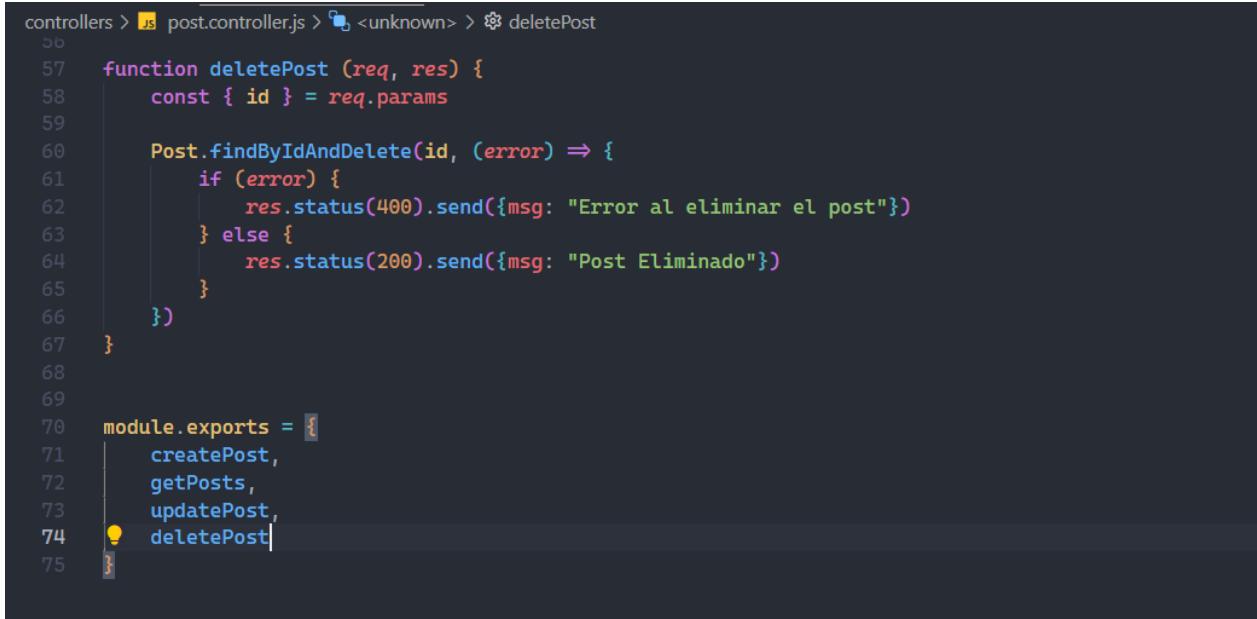
```
1+ {
2+   "docs": [
3+     {
4+       "_id": "652dc5b8db13698df1e368d5",
5+       "title": "Como programar nivel 2",
6+       "content": "El mundo de la programacion 2",
7+       "path": "/como_programar_2",
8+       "created_at": "2023-10-16T23:22:32.225Z",
9+       "miniature": "blog/nlObUyGbgjLeHr8xSM1GiwEP.png",
10+      "__v": 0
11+    },
12+    {
13+      "_id": "652dc45fdb13698df1e368d3",
14+      "title": "Como programar",
15+      "content": "Bienvenidos al mundo de la programacion",
16+      "path": "/como_programar",
17+      "created_at": "2023-10-16T23:16:47.009Z",
18+      "miniature": "blog/l9Y0uYeZlyqVFc1UhLZFcduF.png",
19+      "__v": 0
20+    },
21+    {
22+      "_id": "652dd1b1b9af7f64ed9990d3",
23+      "title": "Como programar nivel 3",
24+      "content": "El mundo de la programacion y algo mas",
25+      "path": "/como_programar_y_algo_mas",
26+      "created_at": null,
27+      "miniature": "blog/FlnWloFRu6gvCrQyPXNTkVKc.png",
28+      "__v": 0
29+    }
30+  ],
31+  "total": 10
32+}
```

Below the preview, there is a placeholder text "Enter a URL and send to get a response" and a note "Select a body type from above to send data in the body of a request".

De esta manera ya dejamos nuestra función lista.

ELIMINAR POST

Vamos a crear nuestra función de eliminación, para ello nos ubicamos en `post.controller.js` y creamos la función `deletePost`:



```
controllers > post.controller.js > deletePost
56
57  function deletePost (req, res) {
58    const { id } = req.params
59
60    Post.findByIdAndDelete(id, (error) => {
61      if (error) {
62        res.status(400).send({msg: "Error al eliminar el post"})
63      } else {
64        res.status(200).send({msg: "Post Eliminado"})
65      }
66    })
67  }
68
69
70  module.exports = {
71    createPost,
72    getPosts,
73    updatePost,
74    deletePost
75  }
```

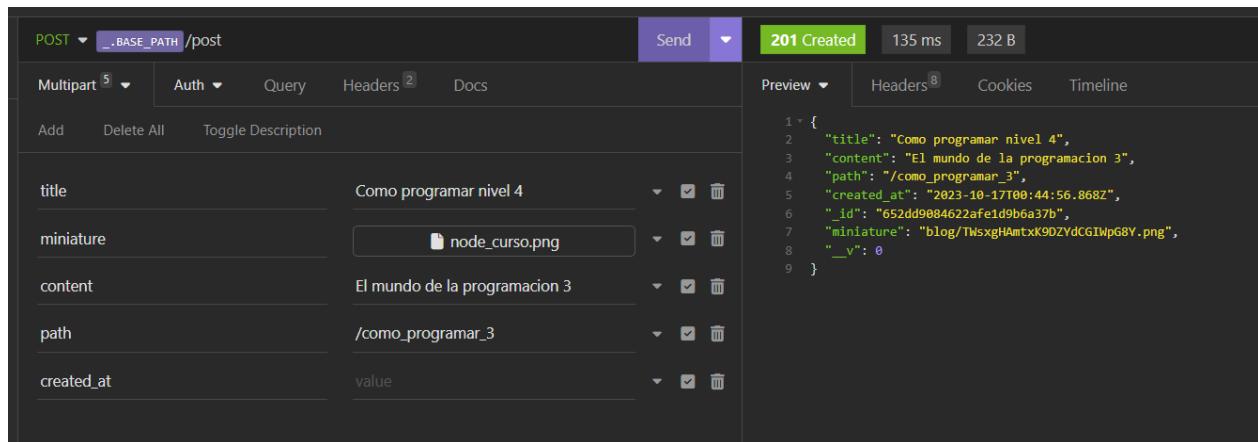


@hdtoledo

Nuestra función es muy similar a la que utilizamos anteriormente, ahora vamos a nuestro post.router.js y creamos la ruta:

```
router > js post.router.js > ...
1 const express = require("express")
2 const multiparty = require("connect-multiparty")
3 const PostController = require("../controllers/post.controller")
4 const md_auth = require("../middlewares/authenticated")
5
6 const md_upload = multiparty({ uploadDir: "./uploads/blog" })
7 const api = express.Router()
8
9
10 api.post("/post", [md_auth.asureAuth, md_upload], PostController.createPost)
11 api.get("/post", PostController.getPosts)
12 api.patch("/post/:id", [md_auth.asureAuth, md_upload], PostController.updatePost)
13 api.delete("/post/:id", [md_auth.asureAuth], PostController.deletePost)
14
15
16 module.exports = api
```

Ahora vamos a crear un post nuevo en insomnia:



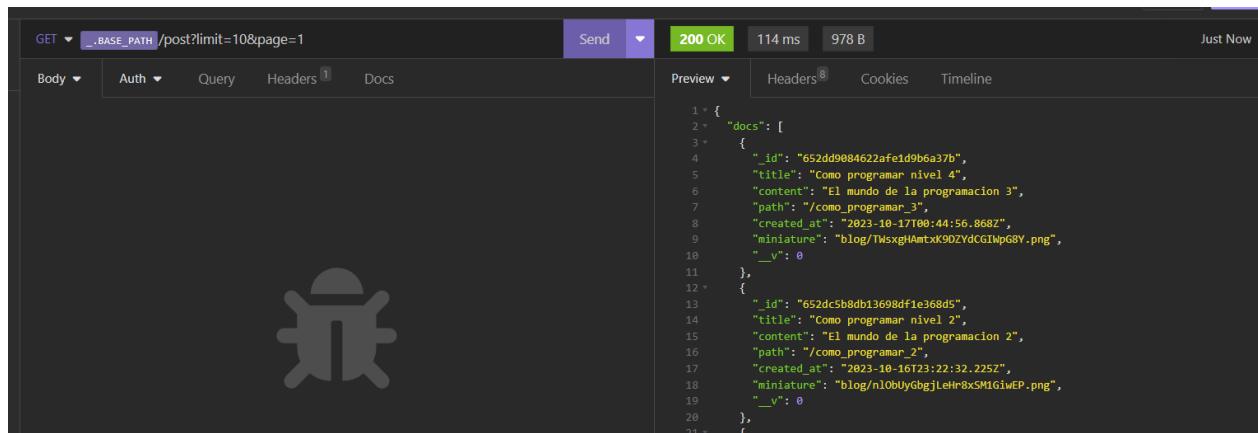
The screenshot shows the insomnia API client interface. A POST request is being made to `.BASE_PATH /post`. The request body contains the following fields:

- title**: Como programar nivel 4
- miniature**: node_curso.png
- content**: El mundo de la programacion 3
- path**: /como_programar_3
- created_at**: value

The response status is **201 Created**, with a response time of 135 ms and a response size of 232 B. The response body is displayed in a code editor-like format:

```
1 < {
2   "title": "Como programar nivel 4",
3   "content": "El mundo de la programacion 3",
4   "path": "/como_programar_3",
5   "created_at": "2023-10-17T00:44:56.868Z",
6   "_id": "652dd9084622afe1d9b6a37b",
7   "miniature": "blog/TNsxgHAmtxK9DZYdCGIwG8Y.png",
8   "__v": 0
9 }
```

Verificamos en `getPost`:

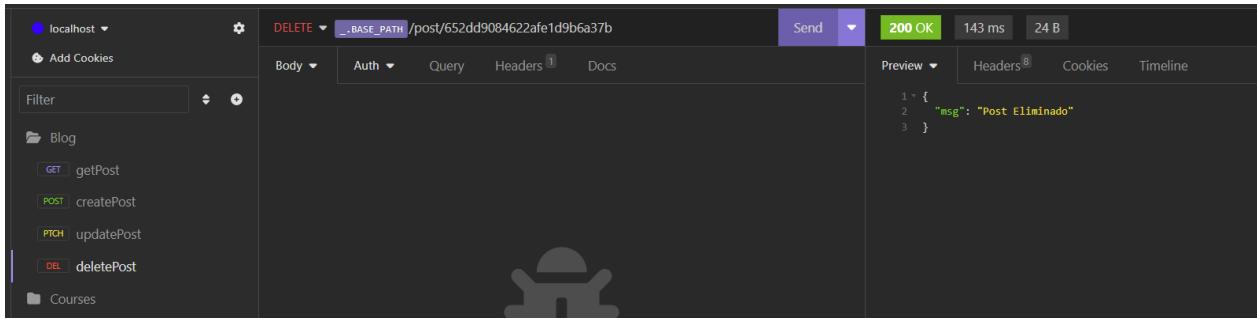


The screenshot shows the insomnia API client interface. A GET request is being made to `.BASE_PATH /post?limit=10&page=1`. The response status is **200 OK**, with a response time of 114 ms and a response size of 978 B. The response body is displayed in a code editor-like format:

```
1 < {
2   "docs": [
3     {
4       "_id": "652dd9084622afe1d9b6a37b",
5       "title": "Como programar nivel 4",
6       "content": "El mundo de la programacion 3",
7       "path": "/como_programar_3",
8       "created_at": "2023-10-17T00:44:56.868Z",
9       "miniature": "blog/TNsxgHAmtxK9DZYdCGIwG8Y.png",
10      "__v": 0
11    },
12    {
13      "_id": "652dc5b8db13698df1e368d5",
14      "title": "Como programar nivel 2",
15      "content": "El mundo de la programacion 2",
16      "path": "/como_programar_2",
17      "created_at": "2023-10-16T23:22:32.225Z",
18      "miniature": "blog/nlObUyGbgjLeHr8xSMIGiWEP.png",
19      "__v": 0
20    }
21 }
```

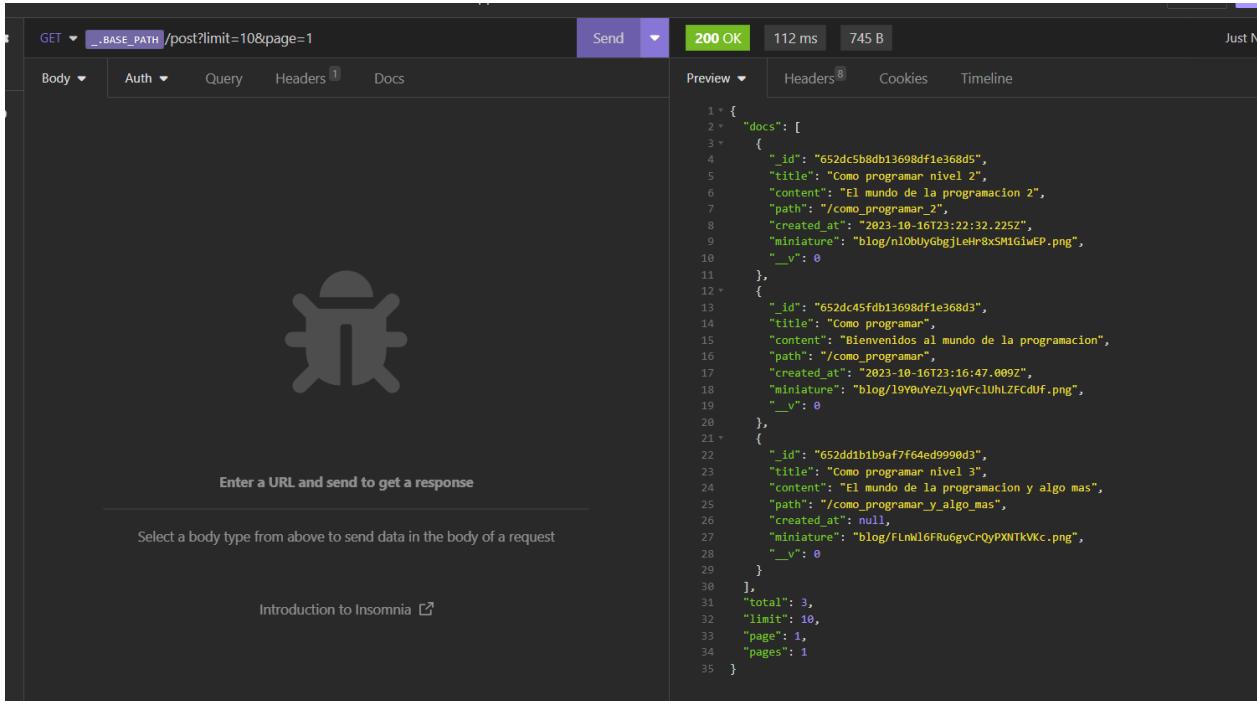


Tomamos el id de nuestro nuevo post y ahora creamos la ruta de delete en nuestro insomnia:



The screenshot shows the Insomnia REST client interface. On the left, there's a sidebar with a 'Blog' section containing 'getPost', 'createPost', 'updatePost', and 'deletePost' endpoints. The main area shows a 'DELETE' request to `__BASE_PATH/post/652dd9084622afe1d9b6a37b`. The response is a 200 OK status with a 'msg' key containing the value 'Post Eliminado'.

De esta manera comprobamos que se ha eliminado nuestro post, vamos a verificar en `getPost`:



The screenshot shows the Insomnia REST client interface. A 'GET' request is made to `__BASE_PATH/post?limit=10&page=1`. The response is a 200 OK status with a JSON payload containing three posts. Each post has fields like '_id', 'title', 'content', 'path', 'created_at', 'miniature', and '_v'. The posts are: 1. 'Como programar nivel 2', 'El mundo de la programacion 2', '/como_programar_2', '2023-10-16T23:22:32.225Z', 'blog/n10blyGhgjLeHr8x5MIGiweP.png', 0. 2. 'Como programar', 'Bienvenidos al mundo de la programacion', '/como_programar', '2023-10-16T23:16:47.009Z', 'blog/l9Y0uYe2LyqVFc1UhLzFCdUf.png', 0. 3. 'Como programar nivel 3', 'El mundo de la programacion y algo mas', '/como_programar_y_algo_mas', null, 'blog/FlnWl6FRu6gvCrQyPXNTkVKc.png', 0.

De esta manera damos por terminado nuestra función delete.



OBTENER UN POST POR SU PATH

Ahora vamos a crear nuestra función para poder obtener el path de un post, para que podamos acceder directamente a ese post en nuestras publicaciones, así que vamos a crear la función `getPost` en singular para poder obtener esa ruta de la publicación:

```
controllers > post.controller.js > ...
...
70  function getPost(req, res) {
71      const { path } = req.params
72
73      Post.findOne({ path }, (error, postStored) => {
74          if (error) {
75              res.status(500).send({msg: "Error del servidor"})
76          } else if (!postStored) {
77              res.status(400).send({msg: "No se ha encontrado ningun post"})
78          } else {
79              res.status(400).send(postStored)
80          }
81      })
82  }
83
84  module.exports = {
85      createPost,
86      getPosts,
87      updatePost,
88      deletePost,
89      getPost
90  }
```

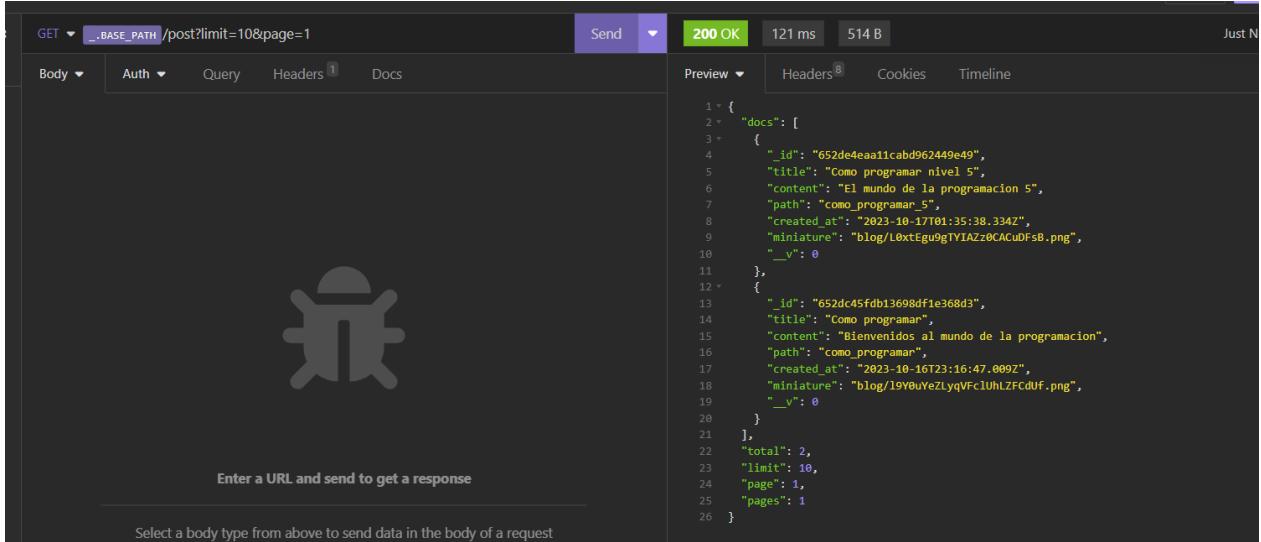
Como podemos observar a través de la función **findOne** obtenemos el path y validamos a través de los diferentes errores que nos puede generar con mensajes, vamos a agregar nuestra ruta en post.router.js:

```
router > post.router.js > ...
1 const express = require("express")
2 const multiparty = require("connect-multiparty")
3 const PostController = require("../controllers/post.controller")
4 const md_auth = require("../middlewares/authenticated")
5
6 const md_upload = multiparty({ uploadDir: "./uploads/blog"})
7 const api = express.Router()
8
9
10 api.post("/post", [md_auth.asureAuth, md_upload], PostController.createPost)
11 api.get("/post", PostController.getPosts)
12 api.patch("/post/:id", [md_auth.asureAuth, md_upload], PostController.updatePost)
13 api.delete("/post/:id", [md_auth.asureAuth], PostController.deletePost)
14 api.get("/post/:path", PostController.getPost)
15
16
17 module.exports = api
```



@hdtoledo

ahora vamos a obtener la ruta de nuestro post a través de getPosts:



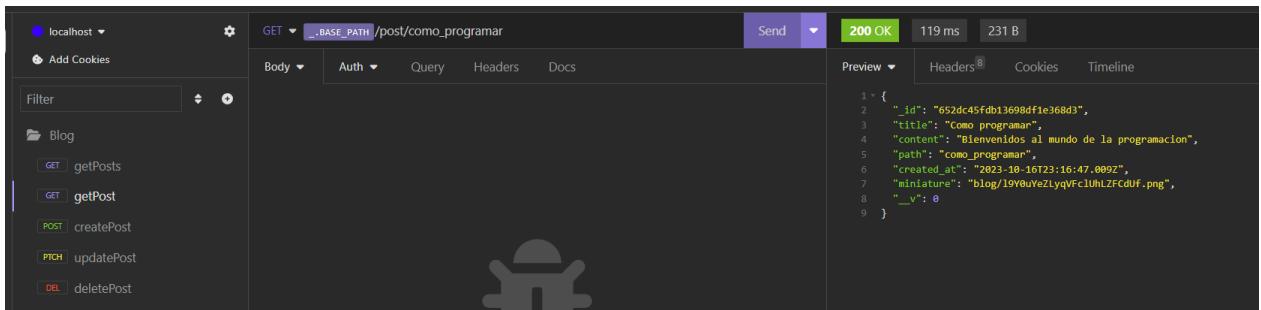
The screenshot shows a Postman interface with the following details:

- Method: GET
- URL: `__BASE_PATH/post?limit=10&page=1`
- Status: 200 OK
- Headers: 8
- Body: Preview (JSON response)
- Timeline: Just Now

The JSON response is as follows:

```
1 + {
2   "docs": [
3     {
4       "_id": "652de4ea11cabd962449e49",
5       "title": "Como programar nivel 5",
6       "content": "El mundo de la programacion 5",
7       "path": "como_programar_5",
8       "created_at": "2023-10-17T01:35:38.334Z",
9       "miniature": "blog/L0xtEgu9gTYIAZz0CACUDFsB.png",
10      "__v": 0
11    },
12    {
13      "_id": "652dc45fdb13698df1e368d3",
14      "title": "Como programar",
15      "content": "Bienvenidos al mundo de la programacion",
16      "path": "como_programar",
17      "created_at": "2023-10-16T23:16:47.009Z",
18      "miniature": "blog/l9v0uYeZlyqVFc1UhLZFCdUf.png",
19      "__v": 0
20    }
21  ],
22  "total": 2,
23  "limit": 10,
24  "page": 1,
25  "pages": 1
26 }
```

Copiamos el path y vamos a crear la ruta de nuestro **getPost** y enviamos la petición:



The screenshot shows a Postman interface with the following details:

- Method: GET
- URL: `__BASE_PATH/post/como_programar`
- Status: 200 OK
- Headers: 8
- Body: Preview (JSON response)
- Timeline: Just Now

The JSON response is as follows:

```
1 + {
2   "_id": "652dc45fdb13698df1e368d3",
3   "title": "Como programar",
4   "content": "Bienvenidos al mundo de la programacion",
5   "path": "como_programar",
6   "created_at": "2023-10-16T23:16:47.009Z",
7   "miniature": "blog/l9v0uYeZlyqVFc1UhLZFCdUf.png",
8   "__v": 0
9 }
```

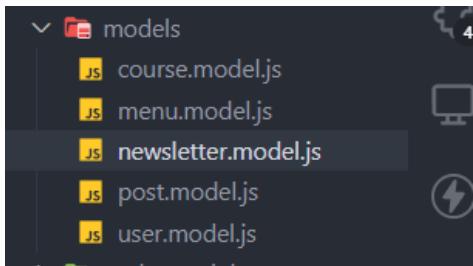
De esta manera vamos a obtener los datos de nuestro post a través del path.



@hdtoledo

MODELO NEWSLETTER

En nuestra aplicación vamos a almacenar los correos de las personas que quieren que les llegue la información relacionada de nuestra web, con todo el contenido, así que vamos a crear nuestro modelo de una manera sencilla, ya que lo único que requerimos es el correo electrónico, vamos a crear en la carpeta **models** nuestro archivo **newsletter.model.js**



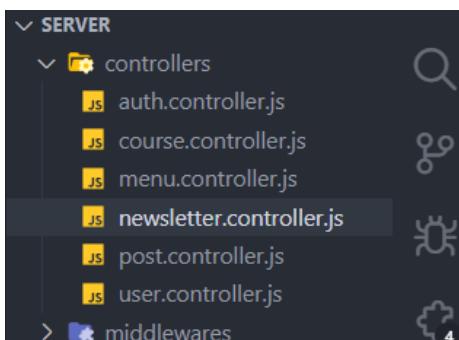
Ahora dentro vamos a dejar la estructura siguiente:

```
models > newsletter.model.js > <unknown>
1  const mongoose = require("mongoose")
2
3  const NewsletterSchema = mongoose.Schema({
4      email: {
5          type: String,
6          unique: true,
7      }
8  })
9
10 module.exports = mongoose.model("Newsletter", NewsletterSchema)
```

De esta manera dejamos nuestro modelo.

ESTRUCTURA API NEWSLETTER

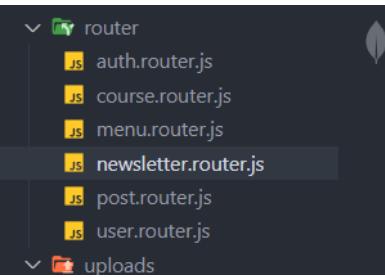
Vamos a crear la estructura de nuestro API newsletter, para ello vamos a crear en **controller** nuestro archivo **newsletter.controller.js**:



Ahora dejamos la siguiente estructura:

```
controllers > newsletter.controller.js > <unknown>
1  const Newsletter = require("../models/newsletter.model")
2
3  //Funciones ...
4
5  module.exports = {
6    |
7  }
```

Creamos nuestro archivo **newsletter.router.js** dentro de **router**:



Ahora dejamos la siguiente estructura:

```
router > newsletter.router.js > ...
1  const express = require("express")
2  const NewsletterController = require("../controllers/newsletter.controller")
3
4  const md_auth = require("../middlewares/authenticated")
5
6  const api = express.Router()
7
8  //Routes
9
10
11 module.exports = api
```

Ahora por último lo añadimos a nuestro **app.js** importamos la ruta:

```
js app.js > newsletterRoutes
9  // Importar rutas
10 const authRoutes = require("./router/auth.router")
11 const userRoutes = require("./router/user.router")
12 const menuRoutes = require("./router/menu.router")
13 const courseRoutes = require("./router/course.router")
14 const postRoutes = require("./router/post.router")
15 const newsletterRoutes = require("./router/newsletter.router")
```



Configuramos nuestra ruta:

```
js app.js > ...
27 // Configurar Rutas
28 app.use('/api/${API_VERSION}', authRoutes)
29 app.use('/api/${API_VERSION}', userRoutes)
30 app.use('/api/${API_VERSION}', menuRoutes)
31 app.use('/api/${API_VERSION}', courseRoutes)
32 app.use('/api/${API_VERSION}', postRoutes)
33 app.use('/api/${API_VERSION}', newsletterRoutes)
34
```

Recordemos que todo debe funcionar correctamente en nuestro servidor:

```
[nodemon] starting `node index.js`
#####
### MERN API REST ###
#####
http://localhost:3000/api/v1
|
```

REGISTRAR EMAIL EN NEWSLETTER

Vamos a realizar la función para poder registrar un correo en nuestro newsletter, para ello vamos a irnos a nuestro **newsletter.controller.js** y creamos la función **suscribeEmail**:

```
controllers > js newsletter.controller.js > ✎ suscribeEmail
1 const Newsletter = require("../models/newsletter.model")
2
3 function suscribeEmail(req, res) {
4   const { email } = req.body
5
6   if (!email) res.status(400).send({ msg: "Email obligatorio" })
7
8   const newsletter = new Newsletter({
9     email: email.toLowerCase(),
10   })
11
12   newsletter.save((error) => {
13     if (error) {
14       res.status(400).send({ msg: "El email ya esta registrado" })
15     } else {
16       res.status(200).send({ msg: "Email registrado con exito" })
17     }
18   })
19 }
20
21 module.exports = {
22   suscribeEmail,
23 }
```

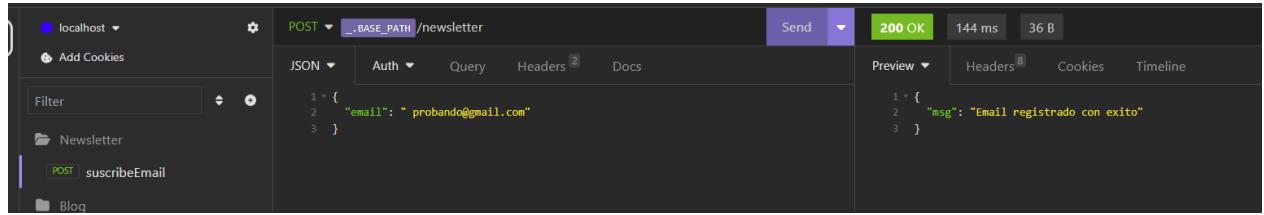


@hdtoledo

Ahora nos vamos a nuestra ruta **newsletter.router.js** y agregamos la ruta:

```
router > js newsletter.router.js > ...
1  const express = require("express")
2  const NewsletterController = require("../controllers/newsletter.controller")
3
4  const md_auth = require("../middlewares/authenticated")
5
6  const api = express.Router()
7
8  api.post("/newsletter", NewsletterController.subscribeEmail)
9
10
11 module.exports = api
```

Ahora en nuestro insomnia vamos a crear una nueva carpeta que se llamará **newsletter** y creamos la nueva petición **subscribeEmail**:



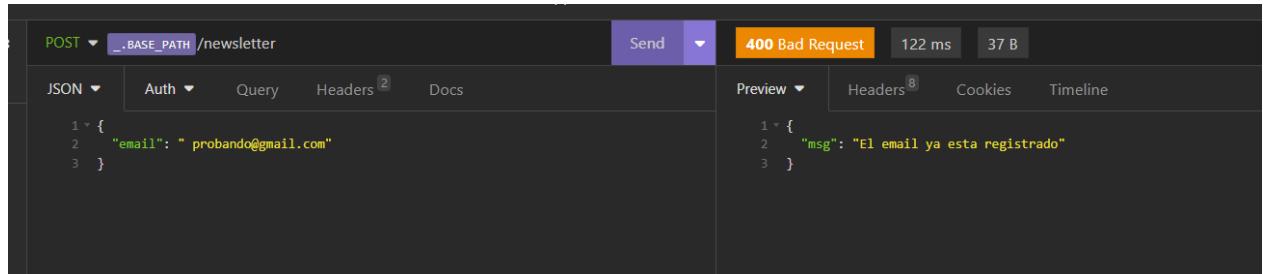
The screenshot shows the insomnia API client interface. A POST request is made to `_.BASE_PATH /newsletter`. The JSON body is `{ "email": "probando@gmail.com" }`. The response is **200 OK** with a response body of `{ "msg": "Email registrado con exito" }`. In the sidebar, there is a tree structure with `Newsletter` expanded, showing the endpoint `subscribeEmail`.

Revisamos en nuestro mongodb:



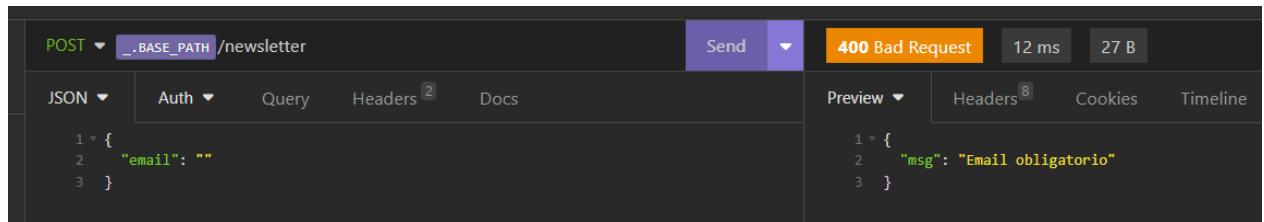
The screenshot shows the MongoDB Compass interface. A query is run against the `newsletters` collection, returning 1 result. The document found is `{_id: ObjectId('...'), email: 'probando@gmail.com', __v: 0}`.

Y si volvemos a intentar hacer la petición con el mismo correo en insomnia:



The screenshot shows the insomnia API client interface. A POST request is made to `_.BASE_PATH /newsletter`. The JSON body is `{ "email": "probando@gmail.com" }`. The response is **400 Bad Request** with a response body of `{ "msg": "El email ya esta registrado" }`. In the sidebar, there is a tree structure with `Newsletter` expanded, showing the endpoint `subscribeEmail`.

Y si lo enviamos sin email en nuestro insomnia:



The screenshot shows the insomnia API client interface. A POST request is made to `_.BASE_PATH /newsletter`. The JSON body is `{ "email": "" }`. The response is **400 Bad Request** with a response body of `{ "msg": "Email obligatorio" }`. In the sidebar, there is a tree structure with `Newsletter` expanded, showing the endpoint `subscribeEmail`.

OBTENER TODOS LOS EMAIL CON PAGINACION

Vamos a realizar una modificación dentro de nuestro modelo para poder hacer la paginación vamos a **newsletter.model.js**:

```
models > JS newsletter.model.js > ...
1  const mongoose = require("mongoose")
2  const mongoosePaginate = require("mongoose-paginate")
3
4  const NewsletterSchema = mongoose.Schema({
5      email: {
6          type: String,
7          unique: true,
8      }
9  })
10
11 NewsletterSchema.plugin(mongoosePaginate)
12
13 module.exports = mongoose.model("Newsletter", NewsletterSchema)
```

Agregamos **mongoose paginate** y lo ponemos en funcionamiento, ahora vamos a crear nuestra función **getEmails** dentro de **newsletter.controller.js**:

```
controllers > JS newsletter.controller.js > 📄 <unknown>
1
2
3
4
5
6
7
8
9
10
11 function getEmails(req, res) {
12     const { page = 1, limit = 10 } = req.query;
13
14     const options = {
15         page: parseInt(page),
16         limit: parseInt(limit),
17     }
18
19     Newsletter.paginate({}, options, (error, emailsStored) => {
20         if (error) {
21             res.status(400).send({ msg: "Error al obtener los emails" })
22         } else {
23             res.status(200).send(emailsStored)
24         }
25     })
26 }
27
28
29
30
31
32
33
34
35
36
37
38
39 module.exports = {
40     suscribeEmail,
41     getEmails,
42 }
```

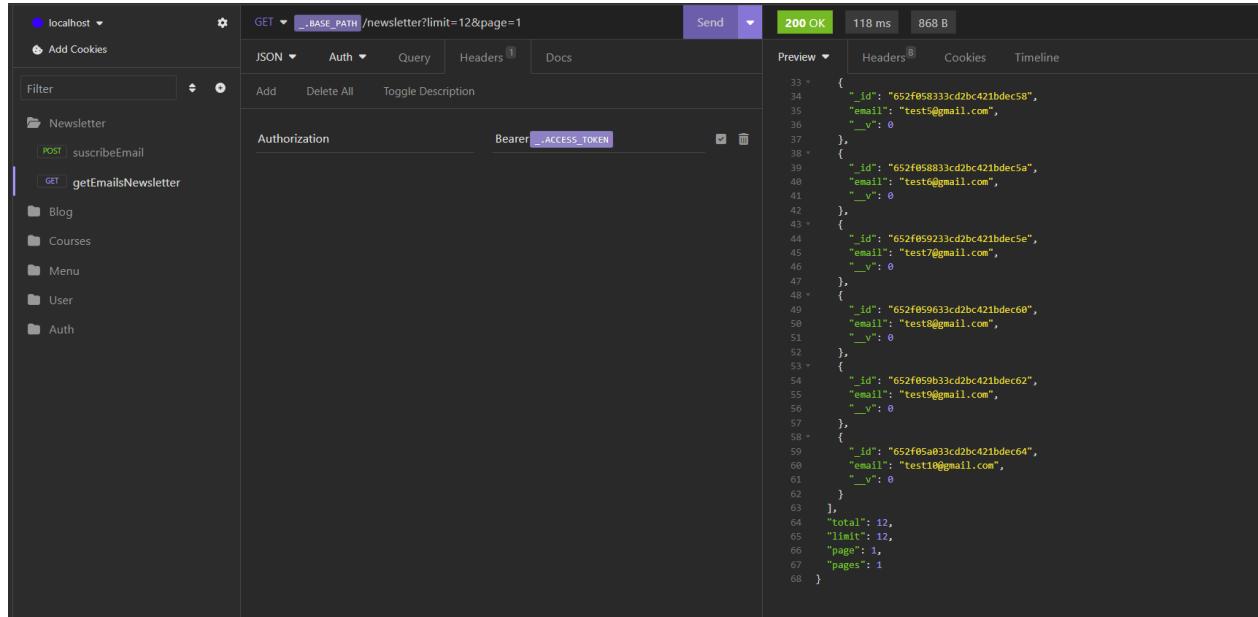


@hdtoledo

Esta función es muy similar a la utilizada anteriormente, ahora vamos a crear la ruta de nuestros mails en `newsletter.router.js`:

```
router > js newsletter.router.js > ...
1 const express = require("express")
2 const NewsletterController = require("../controllers/newsletter.controller")
3 const md_auth = require("../middlewares/authenticated")
4
5 const api = express.Router()
6
7 api.post("/newsletter", NewsletterController.subscribeEmail)
8 api.get("/newsletter", [ md_auth.ensureAuth ], NewsletterController.getEmails)
9
10
11 module.exports = api
```

Recordemos que esta ruta es get y que solo las personas autenticadas podrán acceder a ella, ahora vamos a crear en nuestro insomnia la ruta para poder obtener los correos:



The screenshot shows the Insomnia REST client interface. On the left, there's a sidebar with a tree view of API endpoints under 'Newsletter': `subscribeEmail` (POST), `getEmailsNewsletter` (GET), `Blog`, `Courses`, `Menu`, `User`, and `Auth`. The main panel shows a request configuration for a GET request to `/newsletter?limit=12&page=1`. The 'Headers' tab contains an 'Authorization' header with the value `Bearer ACCESS_TOKEN`. The response is shown in a '200 OK' section with a green status bar. The 'Preview' tab displays the JSON response, which is a paginated list of 12 email documents. Each document has fields: `_id`, `email`, and `_v`. The JSON starts with:

```
33 +
34 {
35   "id": "652f05833cd2bc421bdec58",
36   "email": "test1@gmail.com",
37   "_v": 0
38 },
39 {
40   "id": "652f058833cd2bc421bdec5a",
41   "email": "test6@gmail.com",
42   "_v": 0
43 },
44 {
45   "id": "652f059233cd2bc421bdec5e",
46   "email": "test7@gmail.com",
47   "_v": 0
48 },
49 {
50   "id": "652f059633cd2bc421bdec60",
51   "email": "test8@gmail.com",
52   "_v": 0
53 },
54 {
55   "id": "652f059b33cd2bc421bdec62",
56   "email": "test9@gmail.com",
57   "_v": 0
58 },
59 {
60   "id": "652f05a033cd2bc421bdec64",
61   "email": "test10@gmail.com",
62   "_v": 0
63 },
64 {
65   "total": 12,
66   "limit": 12,
67   "page": 1,
68   "pages": 1
69 }
```

De esta manera ya validamos la obtención de los correos registrados y los tenemos paginados.



@hdtoledo

ELIMINAR UN REGISTRO POR EL EMAIL

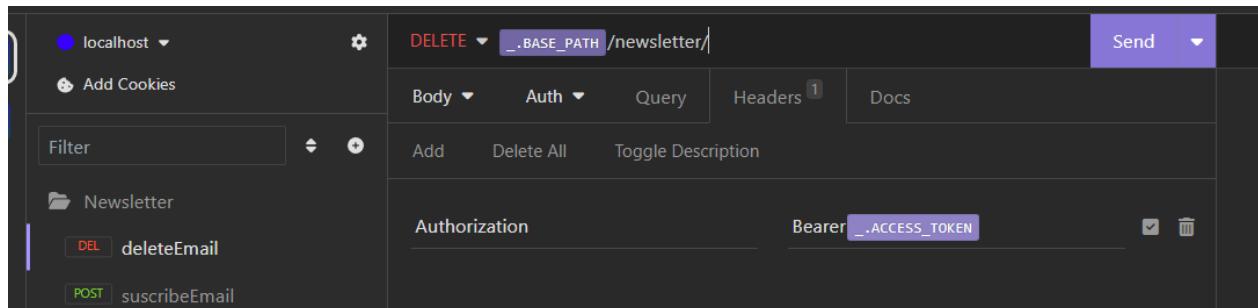
vamos a crear nuestra función **deleteEmail** dentro de **newsletter.controller.js**

```
controllers > JS newsletter.controller.js > <unknown>
38  function deleteEmail(req, res) {
39    const { id } = req.params
40
41    Newsletter.findByIdAndDelete(id, (error) => {
42      if (error) {
43        res.status(400).send({ msg: "Error al eliminar el registro" })
44      } else {
45        res.status(200).send({ msg: "Eliminacion correcta" })
46      }
47    })
48  }
49
50
51  module.exports = [
52    subscribeEmail,
53    getEmails,
54    deleteEmail,
55  ]
```

Nuestra función de eliminación es muy similar a la anterior, ahora vamos a nuestro **newsletter.router.js** y agregamos la ruta **delete**:

```
router > JS newsletter.router.js > ...
1  const express = require("express")
2  const NewsletterController = require("../controllers/newsletter.controller")
3  const md_auth = require("../middlewares/authenticated")
4
5  const api = express.Router()
6
7  api.post("/newsletter", NewsletterController.subscribeEmail)
8  api.get("/newsletter", [ md_auth.asureAuth ], NewsletterController.getEmails)
9  api.delete(["/newsletter/:id"], [ md_auth.asureAuth ], NewsletterController.deleteEmail)
10
11
```

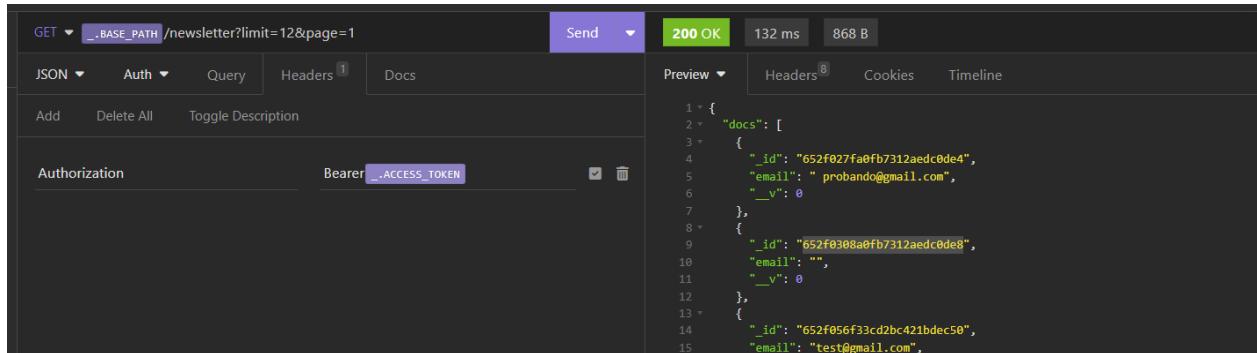
Ahora vamos a crear nuestra ruta **delete** en insomnia con las configuraciones necesarias:



The screenshot shows the insomnia API client interface. On the left, there's a sidebar with a tree view containing a 'Newsletter' folder. Under it, there are two items: 'deleteEmail' (marked with a red 'DEL' icon) and 'subscribeEmail' (marked with a green 'POST' icon). The main panel shows a request configuration for a 'DELETE' operation. The URL is set to '_BASE_PATH /newsletter/:id'. The method dropdown shows 'DELETE'. The body dropdown is set to 'Body'. The auth dropdown is set to 'Auth'. The query dropdown is empty. The headers section has one entry: 'Authorization' with the value 'Bearer _ACCESS_TOKEN'. There are also 'Add', 'Delete All', and 'Toggle Description' buttons. At the bottom right, there's a 'Send' button.



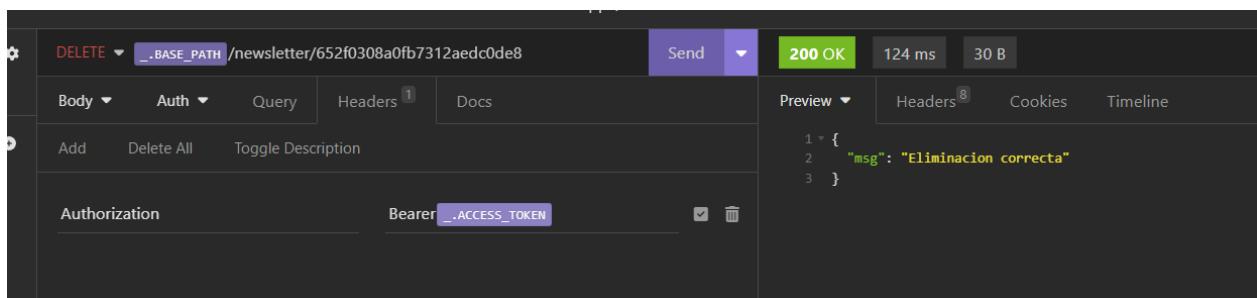
Y vamos a pasarle el id de uno de los correos:



```
1 v {
2 v   "docs": [
3 v     {
4 v       "_id": "652f027fa0fb7312aedc0de4",
5 v       "email": "probando@gmail.com",
6 v       "_v": 0
7 v     },
8 v     {
9 v       "_id": "652f0308a0fb7312aedc0de8",
10 v      "email": "",
11 v      "_v": 0
12 v    },
13 v    {
14 v      "_id": "652f056f33cd2bc421bdec50",
15 v      "email": "test@gmail.com",

```

Seleccionare en este caso el que está sin correo electrónico y lo voy a ejecutar:



```
1 v {
2 v   "msg": "Eliminacion correcta"
3 }
```

De esta manera nos ha quedado configurado nuestra eliminación de correo registrado.



@hdtoledo

CREANDO EL PROYECTO DE REACT

Ya tenemos nuestro **backend** funcionando y ahora llego el turno de crear todo el **frontend** que se encargara de consumir nuestra API.



@hdtoledo