



express

## Introducción a Node.js y Express

### Objetivos:

- Comprender qué es Node.js y por qué es una elección popular para el desarrollo del servidor.
- Familiarizarse con el framework Express.js y su papel en la creación de servidores web.
- Comprender MongoDB y por qué es una elección popular para el alojamiento de información.

### ¿Qué es Node.js?

- Node.js es un entorno de tiempo de ejecución de código abierto que permite ejecutar JavaScript en el lado del servidor. A diferencia de JavaScript tradicional que se ejecuta en el navegador, Node.js posibilita la creación de aplicaciones de servidor altamente eficientes. Su capacidad para ejecutar JavaScript en el servidor ha revolucionado el desarrollo web y la creación de aplicaciones en tiempo real.
- Node.js se destaca por su arquitectura basada en eventos y su bucle de eventos. En lugar de procesos pesados y bloqueantes, Node.js utiliza un modelo de operaciones no bloqueantes y asíncronas. Esto significa que puede manejar múltiples solicitudes y eventos al mismo tiempo sin que un proceso deba esperar a que se complete una tarea antes de continuar con la siguiente.
- Esta arquitectura hace que Node.js sea particularmente eficiente para aplicaciones web en tiempo real, como chats, juegos en línea y aplicaciones de transmisión en vivo. Puede gestionar muchas conexiones simultáneas sin consumir muchos recursos del servidor. Además, Node.js es compatible con bibliotecas y módulos que facilitan la construcción de estas aplicaciones de manera rápida y eficiente.

### Ventajas de Node.js:

- Lenguaje unificado (JavaScript): Node.js utiliza JavaScript tanto en el lado del cliente como en el del servidor, lo que facilita la transición y el intercambio de código entre el navegador y el servidor. Esto simplifica el desarrollo y mantiene un lenguaje consistente en toda la aplicación.
- Alta velocidad y eficiencia: Gracias a su modelo sin bloqueo y orientado a eventos, Node.js es excepcionalmente rápido y eficiente en la gestión de múltiples solicitudes y conexiones simultáneas. Esto lo convierte en una elección poderosa para aplicaciones que requieren alta capacidad de respuesta y rendimiento, como aplicaciones web en tiempo real.
- Gran comunidad y ecosistema de paquetes (NPM): Node.js cuenta con una comunidad activa de desarrolladores y un extenso repositorio de paquetes en NPM (Node Package Manager). Esto significa que los desarrolladores pueden acceder a una amplia gama de bibliotecas y módulos predefinidos para acelerar el desarrollo y resolver problemas comunes de manera más eficaz.

## ¿Qué es Express.js?

Express.js es un popular y versátil framework web que se utiliza en conjunto con Node.js para simplificar la creación de servidores web y el desarrollo de aplicaciones web. Es conocido por su enfoque minimalista y su flexibilidad, lo que lo convierte en una elección común para desarrolladores que desean construir aplicaciones web de manera eficiente.

### Simplificación de la creación de servidores web, manejo de rutas y middleware


Express.js simplifica significativamente el proceso de creación de servidores web en Node.js. Proporciona una variedad de funciones y utilidades que hacen que la configuración inicial de un servidor sea rápida y sencilla. Además, Express ofrece una forma intuitiva de manejar las rutas de una aplicación. Los desarrolladores pueden definir fácilmente rutas y asociar funciones de control a esas rutas, lo que facilita el enrutamiento y el manejo de las solicitudes del cliente.

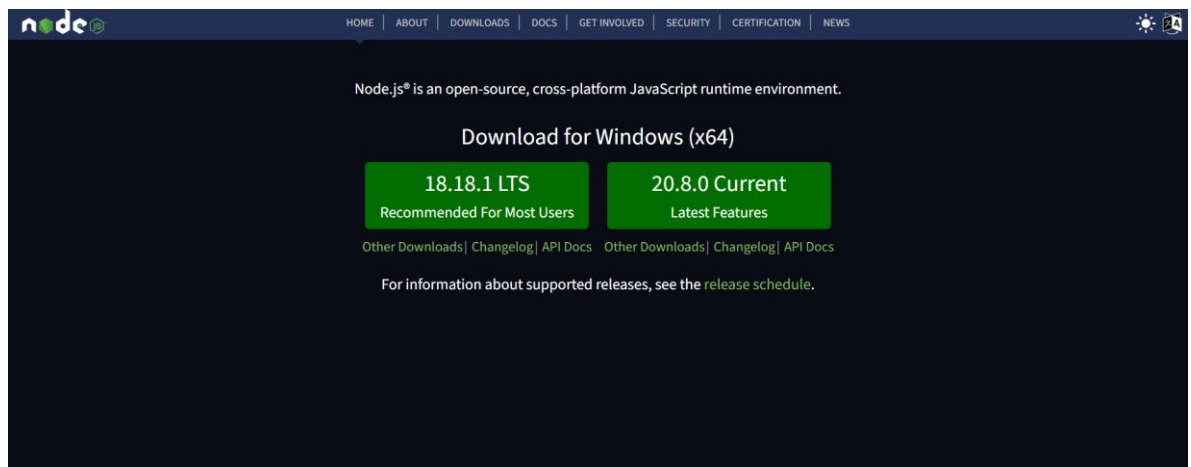
Otra característica poderosa de Express es su soporte para middleware. Los middlewares son funciones que se pueden ejecutar antes o después del manejo de una solicitud. Esto es útil para realizar tareas como autenticación, compresión de datos, registro de solicitudes, entre otras. Express simplifica la incorporación de middleware en la aplicación, lo que permite a los desarrolladores extender y personalizar fácilmente la funcionalidad de sus servidores web.

### Instalación de Node.js y NPM

Node.js se instala junto con NPM (Node Package Manager), que es una herramienta para gestionar paquetes de JavaScript. Sigue estos pasos para instalar Node.js y NPM en Windows, macOS y Linux:

Para Windows:

 Ve al sitio web oficial de Node.js en <https://nodejs.org/>.



- 📁 Descarga el instalador LTS (Long Term Support) para Windows.
- 📁 Ejecuta el instalador y sigue las instrucciones en pantalla. Asegúrate de habilitar la opción "Automatically install the necessary tools..." durante la instalación.
- 📁 Después de la instalación, abre el "Command Prompt" o "PowerShell" y verifica que Node.js y NPM se instalaron correctamente ejecutando los siguientes comandos

```
C:\Users\Fabio>node -v
v20.8.0

C:\Users\Fabio>npm -v
10.2.0

C:\Users\Fabio>|
```

***Nota: Para la instalación de Nodejs, solo elige la ruta donde quieres tener los archivos, lo demás es siguiente siguiente***

## ¿Qué es mongoDD?

MongoDB es una base de datos NoSQL (No solo SQL) de código abierto que se caracteriza por ser una base de datos orientada a documentos. En lugar de almacenar datos en tablas como lo hace una base de datos relacional, MongoDB almacena datos en documentos BSON (Binary JSON) que pueden contener datos con estructuras flexibles y esquemas dinámicos. MongoDB es una base de datos altamente escalable y versátil que se utiliza comúnmente en aplicaciones web, aplicaciones móviles y otros entornos en los que se necesita un almacenamiento de datos rápido y flexible.



## Introducción

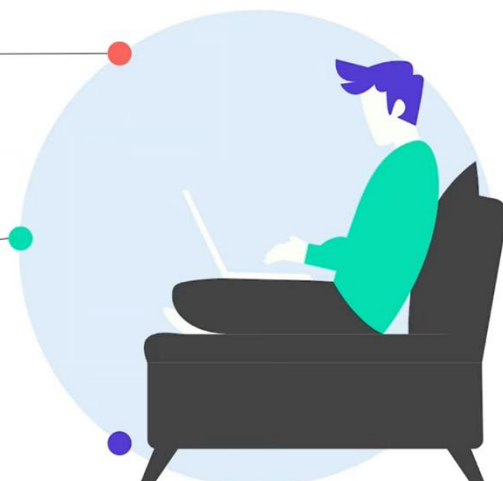
### Qué es MongoDB

MongoDB es una base de datos NoSQL orientada a documentos. Almacena datos en un tipo de formato JSON llamado BSON.



Fundada el año 2007 por Dwight Merriman, Eliot Horowitz y Kevin Ryan, el equipo detrás de **DoubleClick**.

La empresa tenía problemas de escalabilidad y agilidad en un producto que publicaba **400.000 anuncios por segundo**.





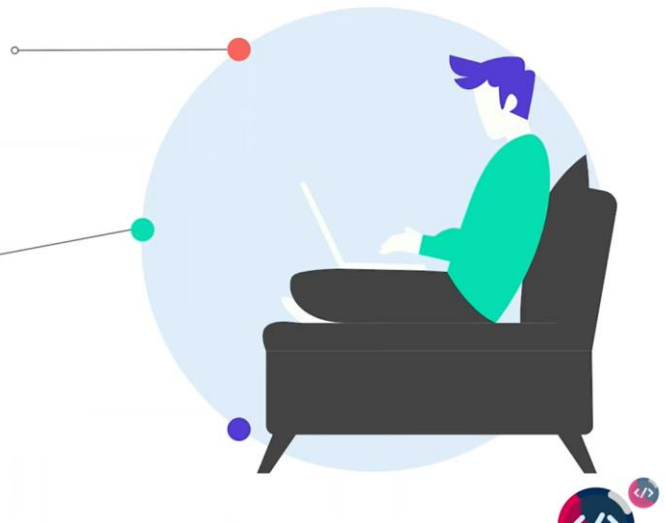
## ¿Cómo se almacenan los datos?

```
{
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
}

{
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
  likes: 1,
  tags: ["news", "events"],
  date: Date()
}
```

MongoDB almacena datos en documentos flexibles JSON/BSON

**MQL** ya contiene múltiples métodos definidos que nos ayudarán a manipular nuestra base de datos.



## Tipos de datos BSON

Según lo comentado anteriormente, una estructura BSON está conformada por clave valor, en donde la clave debe ser del tipo String mientras que el valor puede ser uno de los siguientes tipos:

- Strings.
- Enteros de 32 o 64 bits.
- Tipo de dato real de 64 bits IEEE 754.
- Fecha (número entero de milisegundos en Tiempo Unix).
- Array de bytes (datos binarios).
- booleanos.
- Nulo (null).
- Objetos anidados BSON.
- Array BSON.
- Expresiones Regulares.

## Cómo instalar mongoDB

Visita el sitio web oficial de MongoDB

(<https://www.mongodb.com/try/download/community>)

Selecciona la versión de MongoDB Community Server adecuada para tu sistema operativo;  
En nuestro caso será CommunitySerer

### Community Server

The Community version of our distributed database offers a flexible document data model along with support for:

- Ad-hoc queries
- Secondary indexing
- Real-time aggregations to provide powerul ways to access and analyze your data

Select package

Version

7.0.2 (current)



Platform

Windows x64



Package

msi



Download



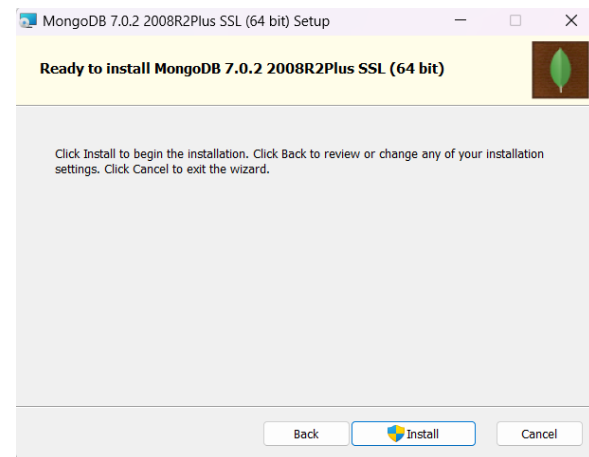
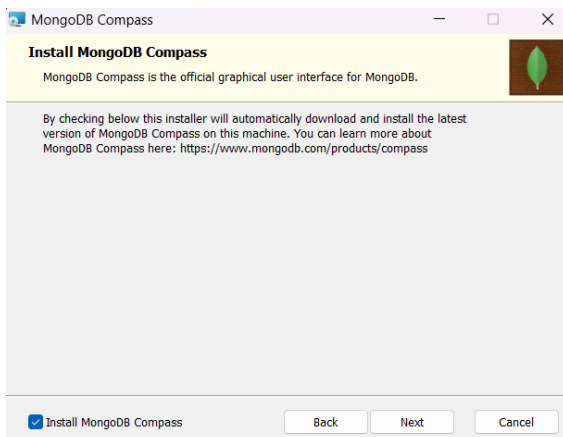
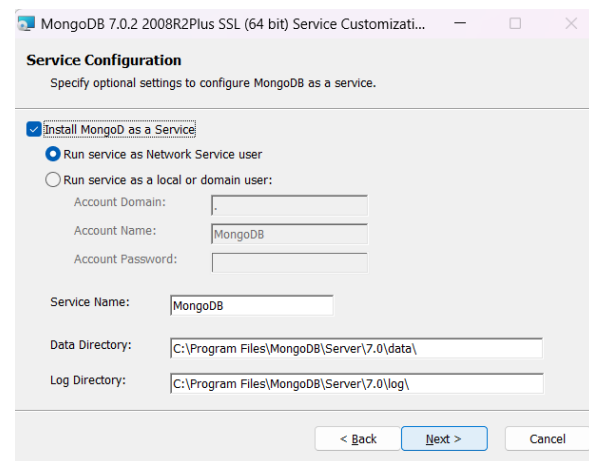
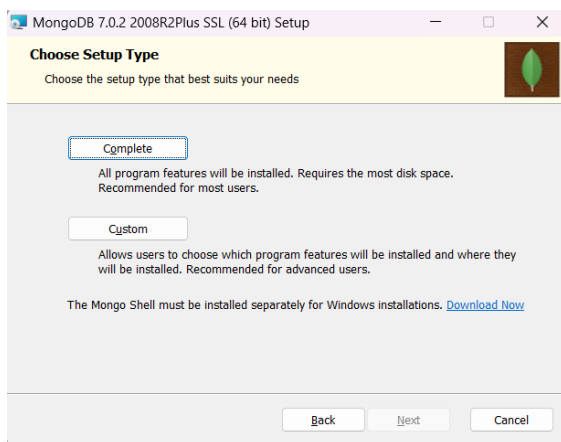
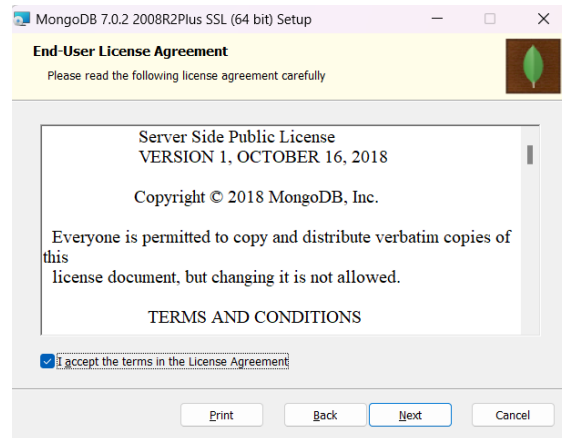
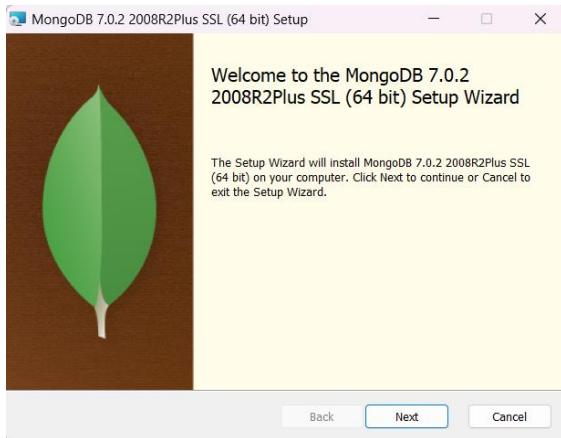
Copy link

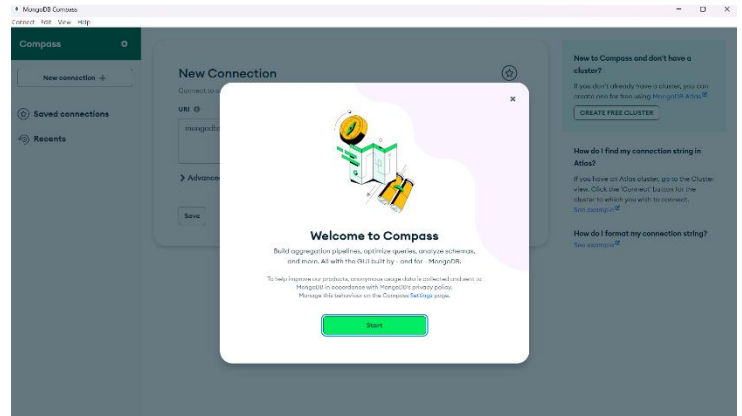
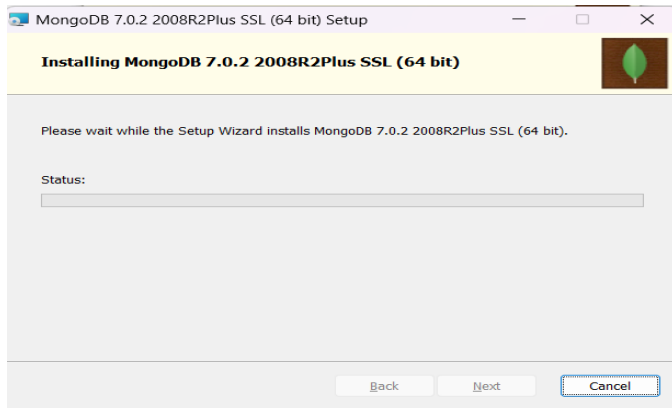
More Options



*Nota: Seleccionamos **msi** para descargar el instalador de instalación de forma directa*

## Pasos para instalar mongo





## New Connection

Connect to a MongoDB deployment

URI ⓘ

Edit Connection String ☒

mongodb://localhost:27017/

Advanced Connection Options

Save

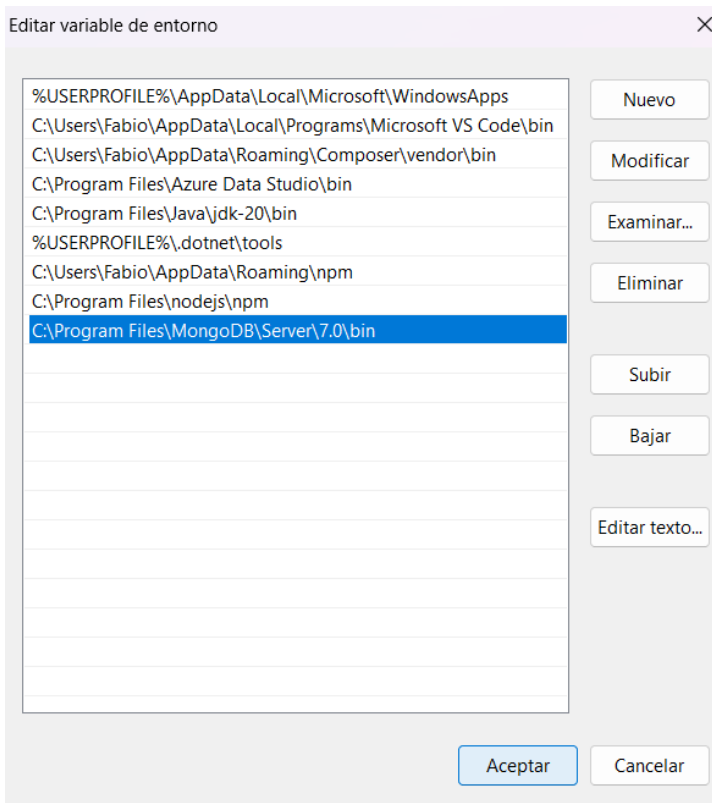
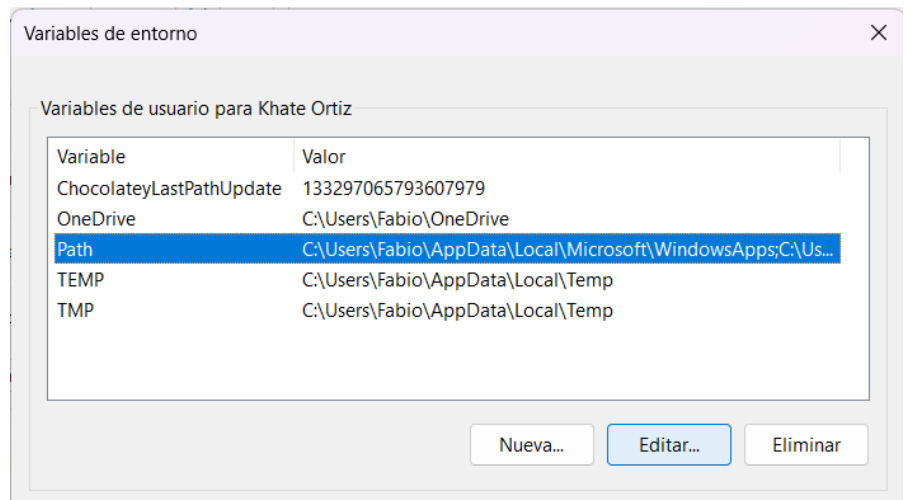
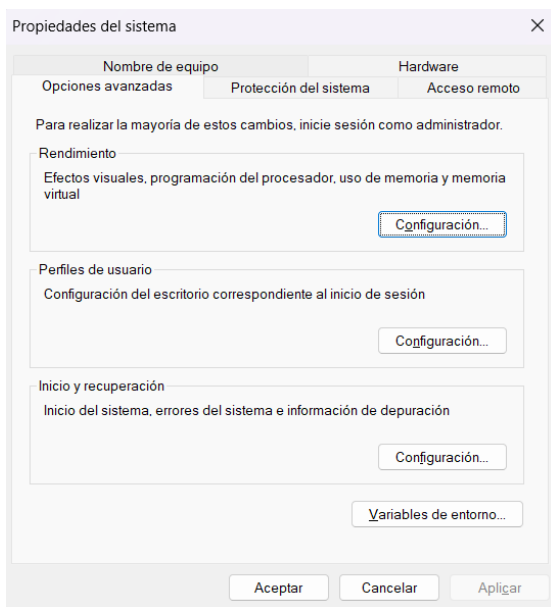
Save & Connect

Connect

Para lograr hacer accesible el servicio de mongoDB desde cualquier parte de nuestro sistema, es necesario verificar que se haya añadido a nuestras variables de entorno la ruta donde quedo nuestro instalador, si lo dejo por defecto en el proceso de instalación debería encontrarse en la siguiente ruta:

**C:\Program Files\MongoDB\Server\7.0\bin**

De lo contrario, se debe buscar la ruta donde se guardó la instalación, así como también verificar si esa ruta esta en nuestras variables de entorno, caso contrario debemos agregarla a nuestro PATH, de la siguiente manera



***Nota: Una vez agregado la variable de entorno a nuestro PATH, damos aceptar a todas las ventanas que se nos crearon.***

Para las personas que no instalaron MongoDB como servicio, para poder conectarse, deberá hacer un segundo paso, en el disco **C** se debe crear una carpeta llamada **data** y dentro de esta carpeta se debe crear otra carpeta llamada **db**, para iniciar el servicio de mongo debe ejecutar en una línea o consola de comandos el siguiente comando **mongod**, esto dará inicio a una instancia que permitirá realizar la conexión a la base de datos

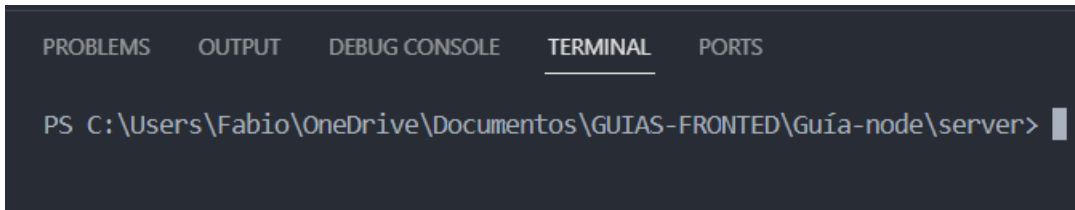
```
C:\Users\Fabio>mongod
```

***Nota: Este comando se debe ejecutar cada vez que se quiera trabajar con mongoDB***

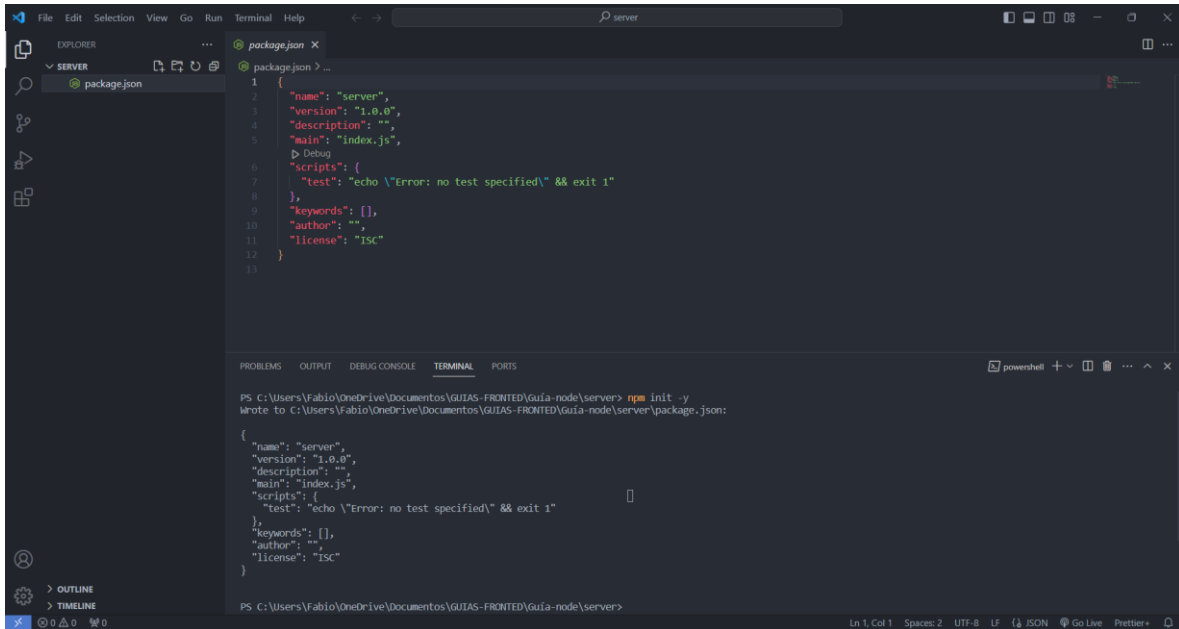


## Creando un proyecto en Nodejs





Creamos una carpeta en donde queremos almacenar nuestro proyecto de node, arrastramos la carpeta dentro de **VsCode**, y una vez estemos en VsCode, abrimos una terminal, para ello podemos usar **ctrl + ñ** (en Windows).



En la consola, ejecutamos el siguiente comando **npm init -y**, este comando nos ayudara a crear el archivo **package.json** el cual se utiliza para poder listar las dependencias que vamos a necesitar



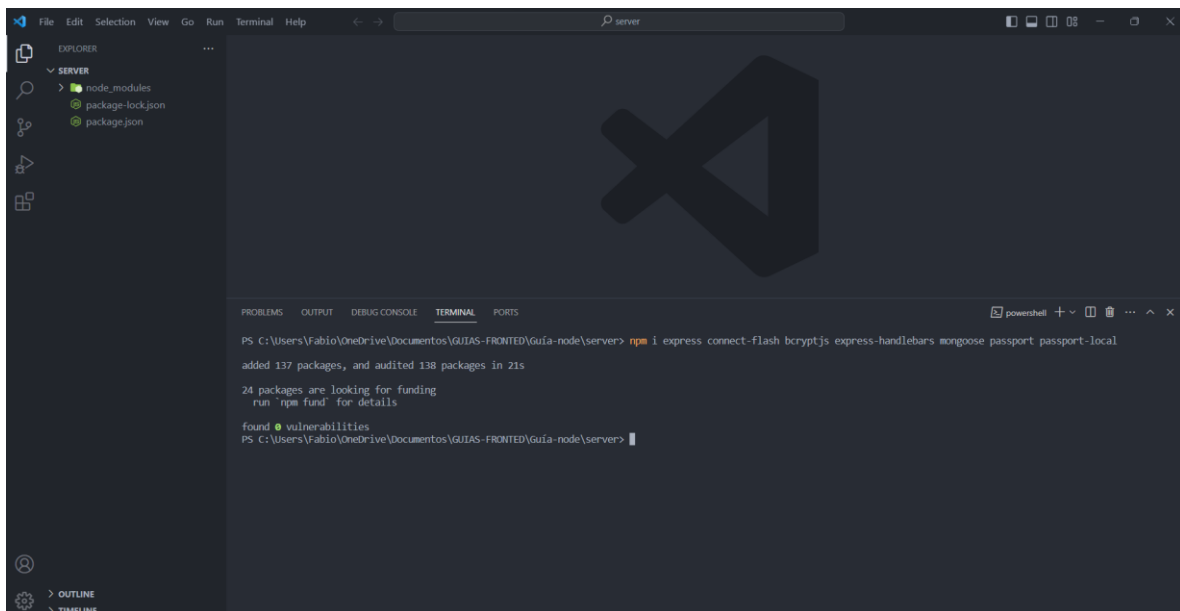
Ahora, vamos a ejecutar un comando que nos sirve para instalar las dependencias que necesitamos para poder iniciar nuestro proyecto CRUD, la cuáles serán las siguientes:

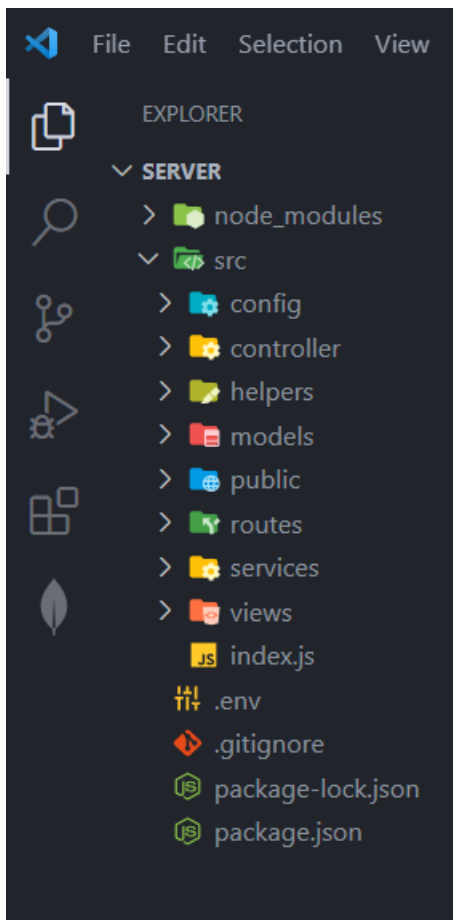
-  **express** (para crear nuestro servidor)
-  **connect-flash** (para poder enviar mensajes al usuario)
-  **bcryptjs** (Para cifrar algún texto, para contraseñas, en hash)
-  **express-handlebars** (Para poder crear vistas desde el servidor html)

- ✚ **express-session** (Para guardar una sesión en el servidor)
- ✚ **method-override** (para poder realizar peticiones HTTPS)
- ✚ **mongoose** (Para poder realizar operaciones con la base de datos de mongoDB)
- ✚ **passport** (Para permitir autenticar usuarios, trabajar en conjunto con bcryptjs)
- ✚ **passport-local** (Para trabajar con autenticación con nuestra base de datos)
- ✚ **nodemon** (Para visualizar los cambios sin tener que recargar el servidor)
- ✚ **dotenv** (para guardar variable de entorno en nuestro proyecto, ya sean contraseñas o cadenas de conexión)
- ✚ **handlebars** (opcional en caso de error con express-handlebars)
- ✚ **morgan** (para ver las peticiones que se están realizando)

**npm i express connect-flash bcryptjs express-handlebars  
mongoose passport passport-local morgan dotenv nodemon**

## Estructura de nuestro proyecto





- node\_modules** -> Es donde se guardan los archivos que node necesita para poder ejecutarse.
- src** -> Es la carpeta principal donde tendremos todo nuestro código.
- confi** -> Carpeta donde contendremos todos nuestros archivos de configuración (base de datos – instancias de servidores, etc...)
- controller** -> Esta carpeta nos ayuda a agrupar toda aquella lógica de exposición de funcionalidad que creemos. (expone nuestras api's)
- models** -> Esta carpeta nos ayuda a agrupar los modelos o clases que representarán la información que deseamos almacenar en nuestra base de datos
- public** -> Sirve para poner cosas que pueden ser accedidas en el proyecto (imágenes, css, archivos, etc...)
- routes** -> Nos ayuda a poder definir las rutas de nuestro proyecto.
- services** -> Buscando una manera de organizar el código, se crea esta carpeta para alojar todo lo que corresponda a la lógica de nuestro negocio y no realizarla directamente en nuestro controller (opcional)
- views** -> En caso de crear un proyecto monolito, sirve para definir las vistas que deseamos renderizar en nuestro proyecto
- index.js** -> Es nuestro archivo principal, donde se iniciará nuestra aplicación
- .env** -> Archivo de configuración local
- package.json** -> Archivo donde estarán alojadas las dependencias que usamos en nuestro desarrollo y producción.

***Nota: La estructura del proyecto varia dependiendo de la definición de cada persona***

## CODIGO DE NUESTRO PROYECTO

**Index.js:** Configuramos nuestro archivo de partida, como deseamos realizar un proyecto modular, tendremos separado en capas toda la lógica de configuración de nuestro servidor y base de datos.

```
1 require('dotenv').config()
2 require('./config/database')
3 const app = require('./config/server')
4
5 app.listen(app.get('port'), () => {
6   console.log("Server on port: " + app.get('port'));
7 })
```

**Server.js:** Creamos nuestro servidor y configuramos temas esenciales como *requerir express, crear el puerto sobre el cual se va a ejecutar nuestro proyecto, configuración de rutas, configuración de nuestras variables de entorno local, configuración de carpetas publicas o de renderización de archivos estáticos*

```
1 //requerimos express
2 const express = require('express')
3 const app = express()
4 const path = require('path')//unir carpetas de nuestro proyecto
5 const morgan = require('morgan')
6
7 //! Settings (configuramos Los modulos)
8 //configuramos el puerto de ejecucion
9 app.set('port', process.env.PORT || 4000)
10 app.set('views', path.join(__dirname, 'views')) // Definimos La ruta donde estarán Las vistas(si se desea devolver algun html)
11
12 //! Midelwares (antes de hacer cualquier accion)T
13 app.use(morgan('dev'))
14
15 //TODO: Para convertir Los datos que nos envíen a un formato json
16 app.use(express.json()) //!requerido
17 app.use(express.urlencoded({ extended: false }))
18
19 //! Routes (para nuestras rutas)
20 app.use(require('../routes/index.routes'))
21 app.use(require('../routes/users.routes'))
22
23 //! Static files
24 app.use(express.static(path.join(__dirname, 'public')))//definimos donde estan nuestros archivos estaticos
25
26 module.exports = app//exportamos app
```

## Database.js

Configuramos nuestra conexión a la base de datos de mongoDB, recordemos que el proyecto esta orientado a buenas prácticas, por lo cual, muchos módulos están separados para lograr mantener un código escalable, en este archivo vamos a crear la cadena de conexión con nuestra base de datos, pero al ser datos sensibles **no** debemos colocarlos directamente, si no, crear las variables de entorno, esto para obtener mayor seguridad con los datos sensibles de nuestro proyecto.

Como usaremos **mongoDB**, debemos requerir **mongoose**, el cual es una dependencia o librería que nos ayuda con el lenguaje de consultas.

```
1 //conexion de La base de datos
2 const mongoose = require('mongoose')
3 const { NOTES_APP_HOST, NOTES_NAME_DATABASE } = process.env
4
5 const MONGO_STRING_CONECTION = `mongodb://${NOTES_APP_HOST}/${NOTES_NAME_DATABASE}`
6 mongoose.connect(MONGO_STRING_CONECTION, {
7   useUnifiedTopology: true,
8   useNewUrlParser: true
9 }).then(() => {
10   console.log("Connected to monogDb successfully");
11 }).catch((e) => console.log("Error al conectarse a mongo: ", e))
```

## .env

```
1 //VARIABLES DE ENTORNO LOCAL
2
3 //Host de nuestro servidor
4 NOTES_APP_HOST = 'localhost'
5
6 // Nombre de nuestra base de datos
7 NOTES_NAME_DATABASE = 'crud_node_expres'
```

Creamos nuestras variables de entorno local, en este caso solo definimos el nombre del servidor y el nombre de la base de datos, pero en este archivo podrías colocar todo lo relacionado con datos sensible, para dar un ejemplo, podrías colocar tu usuario y contraseña de acceso a la base de datos, esto en caso de usar otro tipo de base de datos o en su defecto, que hayas definido credenciales para poder acceder a las bases de datos creadas en mongo

## Routes:

Definimos las rutas que servirán para navegar entre nuestro código, en el caso nuestro, estaremos definiendo las rutas que nos ayudarán a exponer la lógica que queremos ejecutar en el cliente, para ello debemos requerir desde express la función o modulo **Router**, el cual nos ayudará a poder crear y hacer accesibles las rutas de nuestro proyecto.

```
1 const { Router } = require('express') //Importamos Router
2 const router = Router() //Creamos una instancia de la funcion Router
3
4 // Requerimos la logica que definimos en nuestro controlador
5 const { getAllUsers, getSingleUser, createUser, updateUser, deleteUser } = require('../controller/users.controller')
6
7 //Definimos nuestras rutas
8 router.get('/api/v1/users', getAllUsers)
9 router.get('/api/v1/users/:id', getSingleUser)
10
11 router.post('/api/v1/users', createUser)
12 router.put('/api/v1/users/:id', updateUser)
13 router.delete('/api/v1/users/:id', deleteUser)
14
15
16 module.exports = router
```

## Controller.js

En nuestro controlador podemos definir toda la lógica, ya sea lógica de negocio o una simple impresión por consola, dependiendo de lo que se requiera podemos realizar toda la lógica dentro de nuestro controlador, pero se recomienda seguir la programación modular, donde separamos esta lógica en un archivo de servicio o api, en el cual estaremos escribiendo la lógica de acceso a datos y demás lógica de negocio que deseemos implementar, esto es opcional, depende del requerimiento, pero se considera como una buena practica separar todo en servicios.

```

1 // Creamos un objeto el cual se va a llenar con las funciones flecha
2 // que estaremos definiendo
3 const usersController = {}
4
5 const { getAllUsers, createUser } = require('../services/users.service')
6
7 usersController.getAllUsers = async (req, res) => {
8   const users = await getAllUsers()
9   res.json(users)
10 }
11
12 usersController.getSingleUser = (req, res) => {
13   console.log(req.params.id);
14   res.send("Un solo usuario")
15 }
16
17 usersController.createUser = async (req, res) => {
18   return await createUser(req.body);
19 }
20
21 usersController.updateUser = (req, res) => {
22   res.send("Actualizando usuario")
23 }
24
25 usersController.deleteUser = (req, res) => {
26   res.send("Eliminando usuario")
27 }
28
29 module.exports = usersController

```

## service.js

En este modulo vamos a crear la lógica de acceso a datos, realizaremos las operaciones necesarias para poder obtener los registros de nuestra base de datos, poder crear nuevos registros, filtrar y demás operaciones relacionadas con la base de datos, dependiendo de la complejidad de lo requerido podemos crear una capa más donde realicemos operaciones con la información a guardar o listar, esto para poder tener buenas prácticas, sin embargo se debe evaluar la viabilidad de crear distintas capas, dado que, para proyectos que no requieran un nivel complejo de lógica, la capa de servicios es más que suficiente.

```

1 const UserSchema = require('../models/UserModel')
2
3 const getAllUsers = async () => {
4   return await UserSchema.find()
5 }
6
7 const saveUser = async (userData) => {
8   let newUser = new UserSchema(userData)
9   newUser.password = await newUser.encryptPass(newUser.password)
10  return await newUser.save()
11 }
12
13 module.exports = { saveUser, getAllUsers }

```

## Modelo o Schema

Como se menciono en la estructura del proyecto, un modelo o Schema, es aquella estructura que definimos para la creación de nuestros registros en la base de datos, Schema es un concepto de mongoose, el cual quiere dar a entender que, se usa para representar la información que necesitamos almacenar y así mismo poder interactuar con la información, como también se mencionó en la introducción a mongoDB, un modelo o Schema se debe relacionar a una colección o documento.

En un Schema podemos realizar acciones como definir las propiedades de nuestro documento, poder validar los argumentos que consideremos obligatorios, definir el tipo de dato que contendrá cada atributo. Es importante entender que esto varía según nuestras necesidades

```
1 // schema: Define lo que desamos guardar dentro de mongo db
2 // model: Permite crear una clase a partir de un Schema para poder crear clases o propiedades
3 const { Schema, model } = require('mongoose')
4 const bcryptjs = require('bcryptjs')
5
6 const UserSchema = new Schema({
7   name: { type: String, require: [true, 'El nombre es requerido'] },
8   age: { type: Number, require: true },
9   phone: { type: Number, require: true },
10  email: { type: String, require: true },
11  username: { type: String, require: true },
12  password: { type: String, require: true },
13 }, {
14   timestamps: true
15 })
16
17 //Para cifrar la contraseña
18 UserSchema.methods.encryptPass = async password => {
19   const salt = await bcryptjs.genSalt(12)//para ver cuantas veces se ejecuta el cifrado
20   return await bcryptjs.hash(password, salt)//ciframos la contraseña
21 }
22
23 //para comparar las contraseñas cifradas
24 UserSchema.methods.matchPass = async function (password) {
25   return await bcryptjs.compare(password, this.password)//esto devuelve true o false
26 }
27
28 module.exports = model('User', UserSchema, 'users')
```

En un Schema podemos crear métodos, lo cuales estarán asociados a su propio contexto, esto quiere decir que podemos crear funciones o métodos los cuales servirán para hacer de nuestro modelo un poco más complejo según el requerimiento que se nos presente

## Conceptos:

Node.js es un entorno de tiempo de ejecución de JavaScript que te permite ejecutar código JavaScript en el servidor. Es especialmente útil para crear aplicaciones web y servicios de backend, ya que permite la manipulación de archivos, acceso a bases de datos y la creación de servidores web.



## Promesas:

En JavaScript, una promesa es un objeto que representa un valor que puede estar disponible en el futuro, o una operación asíncronica que se está realizando. Las promesas tienen tres estados: pendiente (pending), resuelta (fulfilled) o rechazada (rejected). Puedes encadenar múltiples operaciones asíncronicas y manejar el resultado una vez que la promesa se resuelve o rechaza.

```
1  const miPromesa = new Promise((resolve, reject) => {
2    // Realizar alguna operación asíncronica aquí
3    if (operacionExitosa) {
4      resolve('Éxito');
5    } else {
6      reject('Error');
7    }
8  });
9
10 miPromesa
11   .then(resultado => {
12     console.log('La promesa se resolvió con:', resultado);
13   })
14   .catch(error => {
15     console.error('La promesa fue rechazada con error:', error);
16   });
```

## Async/Await:

**async** y **await** son características de JavaScript que hacen que trabajar con promesas sea más legible y fácil de manejar. **async** se usa en la definición de funciones para indicar que la función devuelve una promesa. **await** se utiliza dentro de funciones **async** para esperar que una promesa se resuelva antes de continuar con la ejecución del código.

```
1  async function miFuncion() {
2    try {
3      const resultado1 = await funcionAsincronica1();
4      const resultado2 = await funcionAsincronica2();
5      console.log('Resultado 1:', resultado1);
6      console.log('Resultado 2:', resultado2);
7    } catch (error) {
8      console.error('Ocurrió un error:', error);
9    }
10 }
11
12 miFuncion();
```