



Hechos

QUE

CONECTAN



El futuro digital
es de todos

MinTIC

BACKEND

Universidad
Industrial de
Santander



‘Mision
TIC 2022’

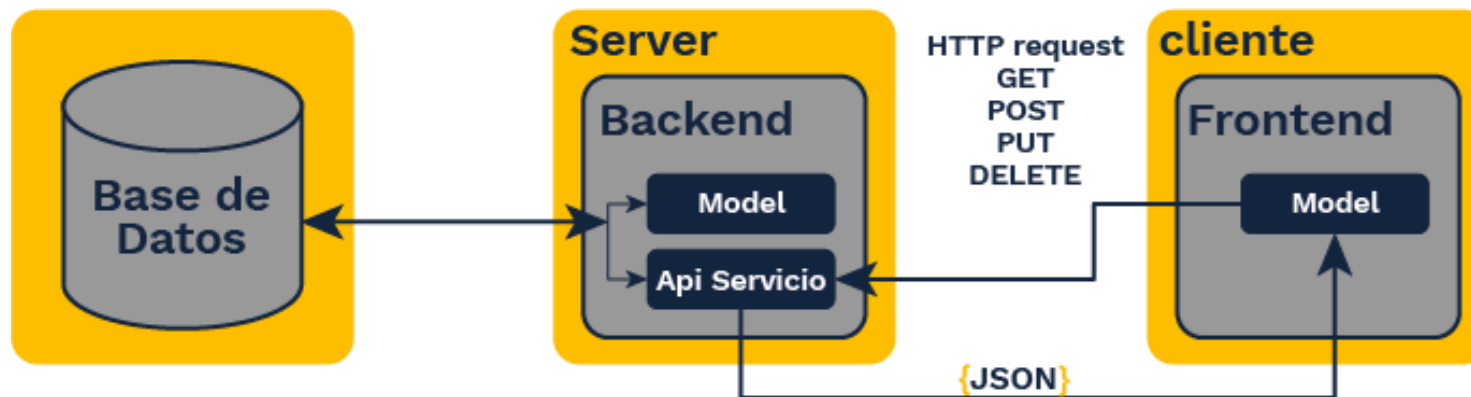
2.1. Manual de creación del proyecto NodeJS

Instalación de nodejs

Para el proceso de instalación podemos guiarnos del video a continuación:

Además, aprenderemos cómo crear el proyecto Backend utilizando NodeJS a través de este video: [Crear proyecto base.](#)

Ahora, vamos a mirar la arquitectura que vamos a tomar en nuestro sistema SPA:



Paso 1. Crear un directorio en el repositorio donde está llevando el proyecto

Consejos para un nombre de proyecto:

- Utiliza siempre minúsculas.
- No utilices espacios en el nombre. Usa guiones mejor.
- Evita el uso de caracteres especiales, signos de puntuación, etc.

Paso 2. La carpeta creada ábrela con un editor de código; aconsejamos que sea Vscode (Visual Studio Code) por su versatilidad para la creación y trabajo de este tipo de desarrollos.

Paso 3. Abre una terminal en Vscode, puedes hacerlo de manera integrada mediante la opción terminal y ejecutas.

npm init

Nos preguntará diferentes opciones, una es el punto de entrada; en este caso, debemos colocar index.js

esto nos crea un archivo **package.json** el cual debe tener más o menos esta forma

```
"name": "backend-project",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

  "test": "echo \"Error: no test specified\" && exit 1"

},

"repository": {

  "type": "git",

  "url": "git+https://github.com/NombreDev/test.git"

},
```

```
"repository": {  
  "type": "git",  
  "url": "git+https://github.com/NombreDev/test.git"  
},  
"keywords": [],  
"author": "",  
"license": "ISC",  
"bugs": {  
  "url": "https://github.com/NombreDev/test/issues"  
},  
"homepage": "https://github.com/NombreDev/test#readme"  
}
```

Ahora estaremos listos para instalar las dependencias.

2.2. Manual para configuración de Dependencias

La configuración de las diferentes dependencias del *Backend*, es un proceso muy importante a la hora de implementar la funcionalidad. En este capítulo, aprenderemos a configurar Express, body-parse, y otros adicionales.

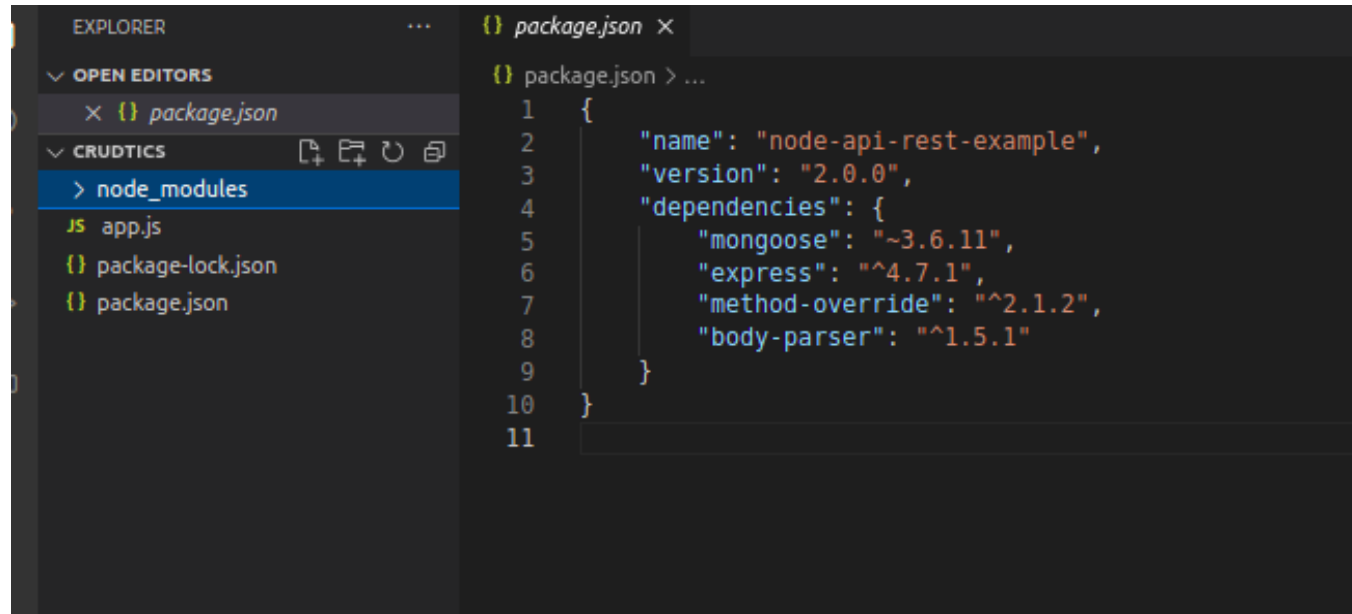
Desde el terminal de VScode ejecutaremos lo siguiente.

Resumen de comandos de Instalación

Express: Express: es el framework que se va a usar durante todo el proceso de construcción del Backend en NodeJS

```
npm install express --save
```

Al ejecutar el comando, se generará la carpeta **node_modules**, en la cual podrás almacenar los módulos necesarios para tu proyecto:



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar is open, showing a project structure with a folder named 'CRUDTICS' containing files 'app.js', 'package-lock.json', and 'package.json'. A new folder named 'node_modules' has been created under 'CRUDTICS'. On the right, the 'package.json' file is open in the editor, displaying the following JSON content:

```
1 {
2   "name": "node-api-rest-example",
3   "version": "2.0.0",
4   "dependencies": {
5     "mongoose": "~3.6.11",
6     "express": "^4.7.1",
7     "method-override": "^2.1.2",
8     "body-parser": "^1.5.1"
9   }
10 }
11
```

bcrypt-nodejs: Dependencia usada en proyectos NodeJS para encriptar información.

```
npm install bcrypt-nodejs --save
```

body-parser: Dependencia usada para parsear las peticiones en formato JSON.

```
npm install body-parser --save
```

connect-multiparty: Dependencia usada para cargar archivos al servidor.

```
npm install connect-multiparty -save
```

jwt-simple: Dependencia usada para la creación de login usando JSON WEB TOKEN.

```
npm install jwt-simple -save
```

moment: Dependencia usada para fechas y tiempo en el Backend.

```
npm install moment -save
```

mongoose: ORM usado en NodeJS para la gestión de bases de datos MongoDB.

```
npm install mongoose -save
```


mongoose-pagination: Dependencia usada en el ORM mongoose para paginar resultados de consultas.

```
npm install mongoose-pagination -save
```

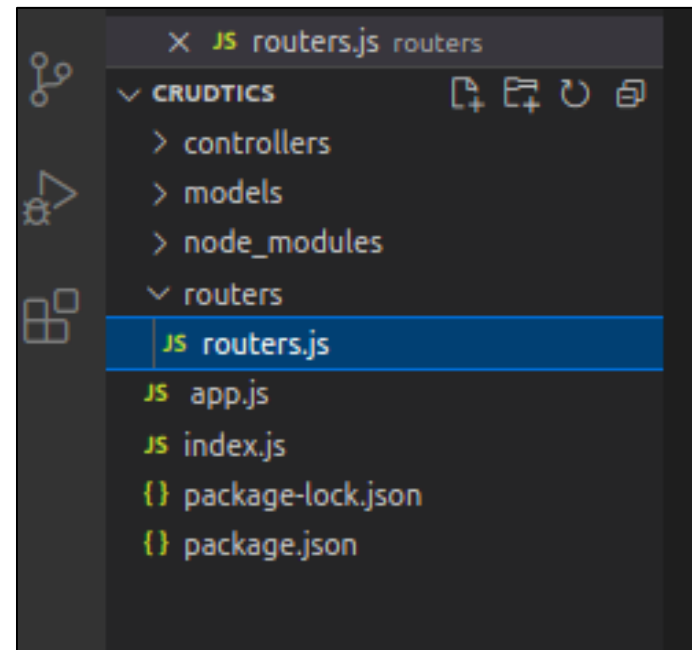
nodemon: Dependencia usada en NodeJs para permitir refrescar el estado del servidor Backend al momento de realizar cambios en la configuración sin necesidad de detener el servicio.

```
npm install nodemon --save-dev
```

El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Instalación dependencia.](#)

2.3. Manual para la creación de la estructura base del Backend

El proceso de desarrollo profesional Web requiere de un orden muy estricto en la estructura del proyecto. El *Backend* es una pieza muy importante en todo el entorno de desarrollo, ya que nos permitirá comunicarnos desde el FrontEnd a la fuente de datos. En esta unidad, crearemos una estructura base para el proyecto. Para ello, podemos crear las carpetas o estructura de nuestro proyecto de la siguiente forma:



- **routers:** En esta se especificarán cada uno de los manejadores de rutas montables y modulares.
- **models:** Especificarán los modelos **donde se trabaja con los datos**, por tanto, contendrá mecanismos para acceder a la información y también para actualizar su estado.
- **controllers:** funciona para obtener los datos solicitados de los modelos y crear la información necesaria para devolvérsela a los clientes en formato json.



Ahora creamos los archivos de **index.js**, el cual será nuestro archivo de punto de entrada inicial, en el cual debe estar el siguiente código:

```
var app = require('./app');  
  
var port = 4000;  
  
app.listen(port, () =>{  
  
    console.log("servidor corriendo ok")  
  
});
```

router.js dentro de la carpeta routers en el cual estarán las rutas de nuestro proyecto

```
const { Router } = require('express');  
  
const router = Router();  
  
export default router;
```

y por último `app.js` en el cual colocaremos las importaciones de nuestro proyecto

```
var express = require("express"),

app = express(),

bodyParser  = require("body-parser"),

methodOverride = require("method-override");

mongoose = require('mongoose');

app.use(express.json());

app.use(express.urlencoded({

    extended: true

}));
```

```
// Configurar cabeceras y cors

app.use((req, res, next) => {

    res.header('Access-Control-Allow-Origin', '*');

    res.header('Access-Control-Allow-Headers', 'Authorization, X-API-KEY, Origin, X-
Requested-With, Content-Type, Accept, Access-Control-Allow-Request-Method');

    res.header('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, DELETE');

    res.header('Allow', 'GET, POST, OPTIONS, PUT, DELETE');

    next();

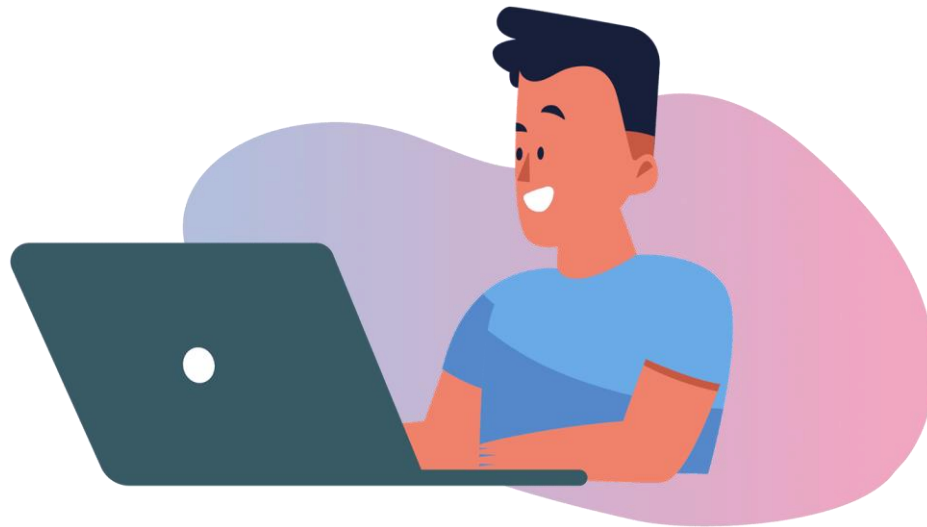
});

module.exports = app;
```

El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Estructura proyecto backend.](#)

2.4. Manual de creación de la Base de Datos

La base de datos es una pieza fundamental en cualquier desarrollo software, ya que en ella se van a almacenar todos los datos de nuestro sistema de información. En este capítulo, revisaremos conceptos de creación de base de datos en MongoDB.



2.4.1. Manual para la creación de colecciones y documentos

En MongoDB, las colecciones son las entidades de nuestra base de datos y los documentos son los registros que almacenamos en ella. En este capítulo, veremos cómo crear las colecciones y documentos en MongoDB.

Paso 1. Primero, entra a la carpeta bin de tu mongodB, posiblemente esté en la ruta

Archivos de Programa>>MongoDB>>Server>>44>bin

Ahí encontraremos las herramientas de administración de MogoDB, seleccionamos **mongo.exe**

Paso 2. Crea la base de datos: lo hacemos con el comando **use** seguido el nombre de la base de datos, terminando con punto y coma.

use nombreBD;

Paso 3. Creación de las colecciones, que es el nombre que se darán a las entidades o tablas en mogodb. cómo lo hacemos bajo la instrucción:

```
db.nombrecoleccion.save({nombreatributo1: valor,nombreatributo2: valor, ... });
```

Para nuestro ejemplo estará, recuerden que los datos se guardan en formato Json.

```
db.municipios.save({nombre: 'Bucaramanga',departamento: '' });
```

paso 4 comprobar la colección guardada con la instrucción

```
db.municipios.find();
```

Esto nos mostrará los datos guardados hasta este momento con el correspondiente id generado por mongodb.

Hasta este momento, lo hemos realizado todo en consola con línea de comandos, pero también lo podemos realizar de manera gráfica con herramientas del mercado, en este caso vamos a usar Robo 3T, que es una herramienta gráfica ligera gratuita para MongoDB que me permite ejecutar consultas, crear índices, visualizar documentos etc.

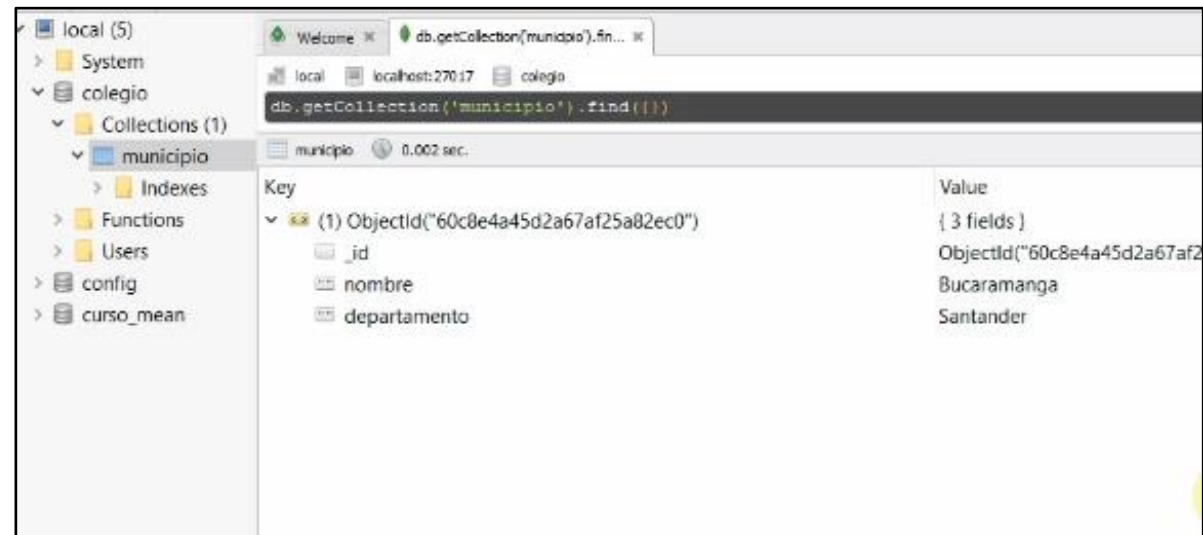
Paso 1. Instala Robo 3T, para ello ve al link <https://robomongo.org/> y selecciona la descarga [Download Studio 3T Only](#) , en la cual encontrarás un formulario pequeño y en el link de descarga, encontraran la version portable o ejecutable, cualquiera de las dos está bien.

Paso 2. Al descargar el programa, lo ejecutan y lo instalan.

Paso 3. Ejecuta Robo 3T y automáticamente el detecta la conexión y la seleccionamos y le damos conectar.

Paso 4. En el navegador de nuestro programa, aparecerá la base de datos, le damos doble clic en Colegio, se nos desplegará un submenú con Collections, Functions, User. Seleccionamos Collections, le damos clic derecho>>create Collection, le damos el nombre, le podemos dar carrera.

Paso 5. Ya con la colección creada, podemos acceder a ella, y con clic derecho, podemos administrar; en estos momentos nos interesa insertar documento que es cada uno de los registros de nuestra entidad.

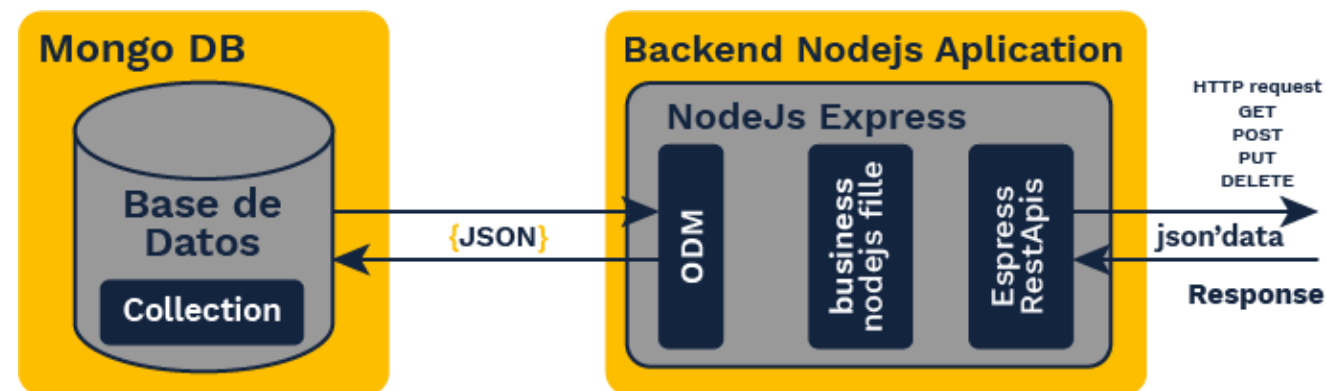


El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Crear BdMongoDB.](#)

2.5. Manual para establecer una conexión a MongoDB desde el Backend

La comunicación del Backend y la base de datos es muy importante, ya que permitirá procesar la información almacenada. En este capítulo, se verá una manera de establecer la comunicación entre la base de datos y el Backend.

Hablemos un poco de la arquitectura propuesta hasta aquí. Tenemos una sistema de bases de datos MongoDB, porque cada vez que conectamos una base no lo hacemos de forma directa, lo hacemos a través de una *data access layer*, que en sí es una interface que permite conectarse de forma transparente para el desarrollador y trae conceptos como persistencia que son importantes para la comunicación con la base de datos.



¿Cómo se conecta NodeJS? lo hace a través del **ODM(Object Document Mapper) Mongoose o de ORM (Object Relational Mapper)** , que nos permite manipular varias bases de datos relacionales de una manera sencilla.

¿Cuál es la diferencia y cómo sabemos cuál usar? Cuando seleccionemos, este proceso se hace dependiendo de la naturaleza del motor de base de datos que se va a utilizar en nuestro proyecto; si usamos de tipo relacional, mysql, postgres, oracle ,etc. Usaremos el ORM, porque son de tipo relacional, y si usamos bases de datos documentales como MongoDB, ArangoDB, CouchDB, etc; usaremos ODM.

Mongoose: Biblioteca ODM para MongoDB y Node. js. Gestiona las relaciones entre los datos, proporciona validación de esquemas y se utiliza para traducir entre objetos en código y la representación de esos objetos en MongoDB.

¿Qué ventajas da? Lo que buscamos es poder cambiar el modelo de base de datos y no tener que cambiar el código; para ello solamente se cambia el driver de comunicación y todo funciona de forma transparente para el desarrollador, como ya se mencionó.

Paso 1. Creamos una nueva carpeta **src**, donde descansará todo nuestro código fuente y otra carpeta **conexDB**, en donde descansa la conexión a la base de datos.

Paso 2. Crea un archivo **conn.js** en la carpeta **conexDB**, la cual nos servirá como una librería para reutilizar la conexión a la base de datos; en el cual, se debe colocar primero la definición de la constante de conexión, la cual le decimos que usará mongoose. Paso seguido de esto, la usaremos con el método **connect**, en el cual se dará la dirección del servidor, el puerto y la base de datos. También tendremos un pequeño código para saber si nos conectamos correctamente.

Al finalizar, exportamos el archivo como librería para ser usado en todo el proyecto con la expresión **module.exports**

```
const mongoose=require('mongoose');
```

```
mongoose
```

```
.connect("mongodb://localhost:27017/colegio", {  
  
  useNewUrlParser: true,
```

```
useCreateIndex: true,  
  
useUnifiedTopology: true,  
  
useFindAndModify: false,  
  
, (err, res) => {  
  
    if (err) {  
  
        throw err;  
  
    } else {  
  
        console.log('La conexión a la base de datos fue correcta...')  
  
    }  
  
});  
  
module.exports = mongoose;
```

Ahora, cómo la usamos, en nuestro archivo de entrada index.js debemos agregar lo siguiente:

```
var mongoose =require('./ app/conexBD/conn');
```

La cual nos dirá que use la librería de conexión en nuestro proyecto.

```
index.js > ...
1  var app = require('./app');
2  var mongoose =require('./ app/conexBD/conn');|
3  var port = 4000;
4  app.listen(port, () =>{
5    console.log("servidor corriendo ok")
6  });
```

El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Conexión BdMongo - Node.](#)

2.6. Manual de creación de modelos y rutas del Backend

Los modelos en el Backend son la representación lógica de las entidades (Tablas) que se encuentran en la base de datos y muchas veces tiene una correspondencia con estas. Hacemos un modelo para representar las tablas por medio de clases y los datos por medio de objetos, así podemos manejar los datos en la *backend* de forma transparente de la base de datos, sin necesidad de acceder directamente a ésta y desacoplando los sistemas de datos (MongoDB) y nuestro desarrollo.

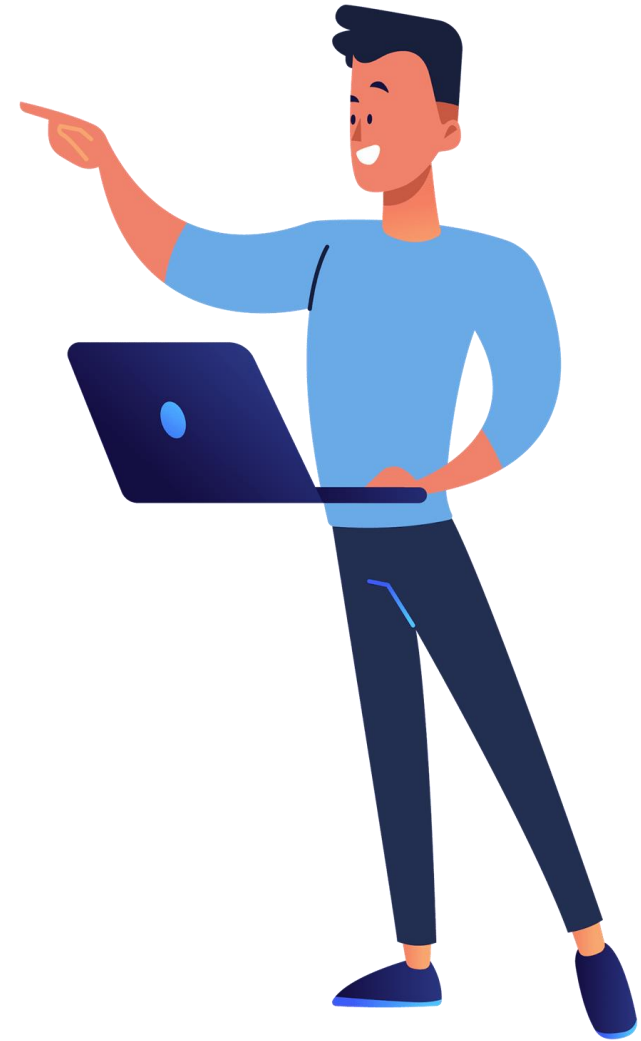


2.6.1. Creación de modelos

Paso 1. Vamos a nuestra carpeta de models y creamos el archivo carrera.js, el cual debe tener lo siguiente: primero la conexión con nuestro ODM, el cual contiene todas la dependencias para crear nuestro modelo **var mongoose=require('mongoose');**

Paso 2. Una constante Schema, donde los modelos se definen mediante la interfaz Schema, la cual permite definir los campos almacenados en cada documento junto con sus requisitos de validación y valores predeterminados:

```
var Schema = mongoose.Schema;
```



Paso 3. Definimos nuestra estructura json, la cual debe contener los atributos de nuestra colección

```
var CarreraSchema=Schema({  
  
  nombre:String,  
  
  escuela:String,  
  
  universidad:String  
  
});
```

Paso 4. Definimos que lo vamos a usar en nuestra aplicación como un módulo.

```
const Carrera = mongoose.model('carrera',CarreraSchema);  
  
module.exports = Carrera;
```

2.6.2. Creación del Controlador

Antes de empezar, vamos a definir que es un **controlador (controller)**, un mediador entre el modelo y los datos que nos pide el cliente; gestionando y adaptando los datos a las necesidades de cada uno, controlando el flujo del mismo..

Para empezar y entender un poco, vamos a hacer una prueba, la cual nos permitirá mirar el funcionamiento y dinámica de nodejs

Paso 1. En nuestra carpeta controllers, creamos una archivo ControllerCarrera.js , el cual tendrá una función de prueba que simplemente nos dará un estado 200 y un mensaje.

```
function prueba(req,res) {  
  
    res.status(200).send({  
  
        message: 'probando una acción'  
  
    });  
  
}
```

Paso 2. Coloca las rutas, que apunten al controlador; para esto, construimos una variable que apunte a nuestro controlador en el archivo `routers>>router.js`

```
var controllerCarrera=require('../controllers/ControllerCarrera');
```

Paso 3. Con la constante `router`, definimos nuestro verbo `get`, el cual lleva la conexión de nuestro api y el controlador que lo define `router.get('/prueba',controllerCarrera);`

Paso 4. Ahora en nuestro archivo `app.js`, importamos las rutas

```
app.use(require('./routers/router'));
```

Paso 5. Corre el servidor y revisa en el explorador con el link <http://localhost:4000/prueba>

Las rutas en el *Backend* son el medio que se utiliza para realizar las diferentes peticiones al Backend. Las peticiones manejan una serie de verbos que van a permitir realizar diferentes tareas en la base de datos. A continuación, se relacionan:

- **POST** : Verbo Http usado para enviar peticiones de inserción de datos.
- **GET** : Verbo Http usado para enviar peticiones de datos. Este verbo es muy usado en selects.
- **DELETE** : Verbo Http usado al momento de requerir eliminar información de la base de datos.
- **PUT**: Verbo http usado al momento de actualizar información de la base de datos.
- **PATCH**: Verbo http usado al momento de requerir cargar archivos al servidor.

```
var mongoose=require('mongoose');

const Schema=mongoose.Schema;

const CarreraSchema=new Schema({

  nombre: String,

  escuela:String,

  universidad:String

});

const Carrera = mongoose.model('carrera', CarreraSchema);

module.exports = Carrera;
```

El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Modelo y Rutas.](#)

2.7. Manual para insertar documentos en colecciones

En estos momentos, ya tenemos el modelo, ahora cómo hacemos los procesos de los requerimientos. Para ello, debemos empezar a descomponerlo en pequeñas acciones para hacerlo general y reutilizable para nuestro proyecto. Para esto utilizamos el CRUD, que es un acrónimo de las cuatro operaciones básicas que realizamos sobre una entidad: **C**reate, **R**ead, **U**ppdate, **D**elete.

CRUD- Operation	SQL	RESTful HTTP	XQuery
Create	INSERT	POST, PUT	insert
Read	SELECT	GET, HEAD	copy/modify/return
Update	UPDATE	PUT, PATCH	replace, rename
Delete	DELETE	DELETE	delete

Paso 1. Definamos el salvar los datos en la entidad que corresponde a Create de nuestro crud. Vamos a nuestro archivo controlador ControllerCarrera.js y digitamos

```
function saveCarrera(req,res) {  
  
    var myCarrera= new Carrera(req.body);  
  
    myCarrera.save((err,result)=>{  
  
        res.status(200).send({message:result});  
  
    });  
  
}
```

Paso 2. Para exponer la función en el mismo archivo debemos adicionar lo siguiente:

```
module.exports={  
  
  prueba,  
  
  saveCarrera  
  
}
```

Paso 3. Exponer la ruta para que el cliente la pueda ver, en el archivo de routes.js

```
router.post('/crear',controllerCarrera.saveCarrera);
```

Paso 4. Probar con un software como Postman o Insomnia, los cuales nos permiten probar la comunicación por el protocolo http.

El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Insert Record Mongo DB.](#)

2.8. Manual para buscar documentos en colecciones

El buscar es una de las partes de las acciones básicas del CRUD, pero para una mayor facilidad, las descomponemos en dos; una que nos pueda mostrar todos los registros, que más adelante podrán paginar, es decir, enviarlas por bloques según un orden específico para cuando son muchos registros, y otra, donde solo necesitamos un registro. La dinámica es la misma, lo único que cambiaremos serán las funciones de controlador.

Paso 1. Definamos el buscar que corresponde a seleccionar un registro de los datos en la entidad correspondiente a Read de nuestro crud. Vamos a nuestro archivo controlador ControllerCarrera.js y digitamos lo siguiente:

```
function buscarData(req, res) {  
  
    var idCarrera=req.params.id;  
  
    Carrera.findById(idCarrera).exec((err, result)=>{  
  
        if(err){  
  
            res.status(500).send({message:'Error al momento de ejecutar la solicitud'});  
  
        }else{  
  
            if(!result){  
  
                res.status(404).send({message:'El registro a buscar no se encuentra disponible'});  
  
            }else{  
  
                res.status(200).send({result});  
  
            }  
  
        }  
  
    });  
  
}
```

Básicamente, sacamos el id o identificador del registro. el cual buscamos en el modelo, para eso usamos la función findById y validamos si está o no existe.

Paso 2. En el mismo controlador, hacemos la función ListarAllData, que es un poco más elaborada, ya que hace dos funciones: la primera es como la que ya realizamos, si el idCarrera es nulo, el result arrojará todos los elementos de nuestra colección ordenados por nombre; de lo contrario, buscará el id , si ocurre algo validará el resultado como la anterior:

```
function listarAllData(req,res){  
  
    var idCarrera=req.params.idb;  
  
    if(!idCarrera){  
  
        var result=Carrera.find({}).sort('nombre');  
  
    }else{  
  
        var result=Carrera.find({_id:idCarrera}).sort('nombre');  
  
    }  
}
```

```
result.exec(function(err,result) {  
  
    if(err) {  
  
        res.status(500).send({message:'Error al momento de ejecutar la solicitud'});  
  
    }else{  
  
        if(!result) {  
  
            res.status(404).send({message:'El registro a buscar no se encuentra disponible'})  
;  
  
        }else{  
  
            res.status(200).send({result});  
  
        }  
  
    }  
  
    })  
  
}
```

Paso 3. Para exponer la función en el mismo archivo, debemos adicionar lo siguiente:

```
module.exports={  
  
  prueba,  
  
  saveCarrera,  
  
  buscarData,  
  
  listarAllData  
  
}
```


Paso 4. Colocar las rutas, primero la de buscar uno en el cual utilizamos `/:id` para designar que tiene un parámetro:

```
router.post('/buscar/:id',controllerCarrera.buscarData);
```

Ahora, colocamos la siguiente ruta que varía un poco, ya que al final del parámetro colocamos `?` para que sea opcional:

```
router.post('/buscar/:id?',controllerCarrera.listarAllData);
```

El detalle de lo explicado anteriormente, se puede observar en el siguiente video: [Consultas Mongo Db](#)

2.9. Manual para actualizar y borrar documentos en colecciones

La actualización tiene dos sentidos: primero tenemos que encontrar el elemento para validar que esté y después actualizarlo, aquí nos valdremos de una función de odm que hace las dos cosas al mismo tiempo **findOneAndUpdate**.

Paso 1. Definamos el actualizar, que corresponde a la actualización de un registro de los datos en la entidad correspondiente a Read de nuestro crud. Luego vamos a nuestro archivo controlador ControllerCarrera.js y digitamos lo siguiente:

```
function updateCarrera(req,res) {  
  
    var id = mongoose.Types.ObjectId(req.query.productId);  
  
    Carrera.findOneAndUpdate({_id: id}, req.body, {new: true}, function(err,  
Carrera) {  
  
        if (err)  
  
            res.send(err);  
  
            res.json(Carrera);  
  
        });  
  
    };  
};
```

Paso 2. Ahora definimos el borrado, el cual es muy parecido a la actualización, porque primero debemos encontrar el registro para poder borrarlo, pero como en la actualización, también podemos hacerlo con un método que hace las dos cosas:

```
function deleteCarrera(req,res){

    var idCarrera=req.params.id;

    Carrera.findByIdAndRemove(id, function(err, carrera){

        if(err) {

            return res.json(500, {

                message: 'No hemos encontrado la carrera'

            })

        }

        return res.json(carrera)

    });

};
```

Paso 3. Exponer la función en el mismo archivo, para ello debemos adicionar lo siguiente:

```
module.exports={  
  
  prueba,  
  
  saveCarrera,  
  
  buscarData,  
  
  listarAllData,  
  
  updateCarrera,  
  
  deleteCarrera  
  
}
```

Paso 4. Colocar las rutas, primero la de buscar uno:

```
router.delete('/carrera/:id',controllerCarrera.deleteCarrera);  
  
router.put('/carrera/:id',controllerCarrera.updateCarrera);
```



El futuro digital
es de todos

MinTIC

Hechos

QUE

CONECTAN

CICLO 4A

EJE TEMÁTICO 2

BACKEND

Universidad
Industrial de
Santander



Mision
TIC 2022