Argentina programa 4.0

Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

primero la gente

UNTREF
UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

CLASE 12: NodeJs MongoDB

Ministerio de
Desarrollo Productivo
Argentina

Agenda de hoy

A. Fundamentos de un driver

B. Instalación de MongoDB (driver oficial en Node.js)

C. Conexión de Node.js con MongoDB

   a. Seguridad al definir una conexión

   b. Acceso al motor/clúster MongoDB

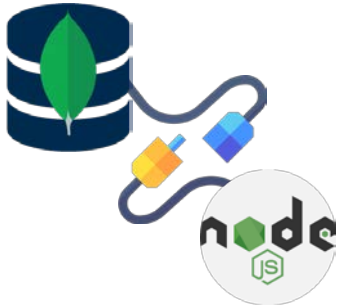
   c. Express JS y MongoDB

   d. Acceso a una bbdd. y a una colección

D. Manejo de códigos de estado y errores

Fundamentos de un Driver

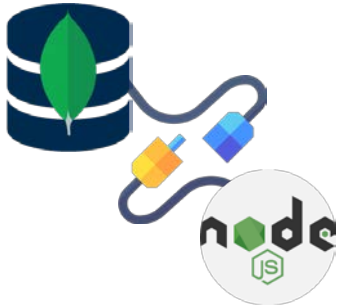
En el contexto de la informática, un controlador (o driver) es un software que permite que un dispositivo o componente se comunique con el sistema operativo u otro software en un equipo.



El controlador es el intermediario entre el hardware y el software, proporcionando una interfaz que el software puede utilizar para interactuar con el dispositivo.

Fundamentos de un Driver

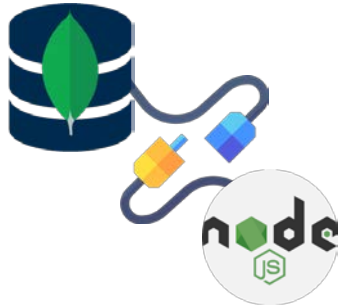
Por ejemplo, un controlador de impresora, escáner, o cualquier otro dispositivo externo a una computadora, permite que el sistema operativo se comunique con la impresora y envíe documentos para imprimir.



Sin el controlador, el sistema operativo no sabría cómo interactuar con la impresora y no se podría imprimir nada.

Fundamentos de un Driver

En el caso del driver oficial de MongoDB para Node.js, este nos permite que las aplicaciones Node.js se comuniquen con una base de datos MongoDB.

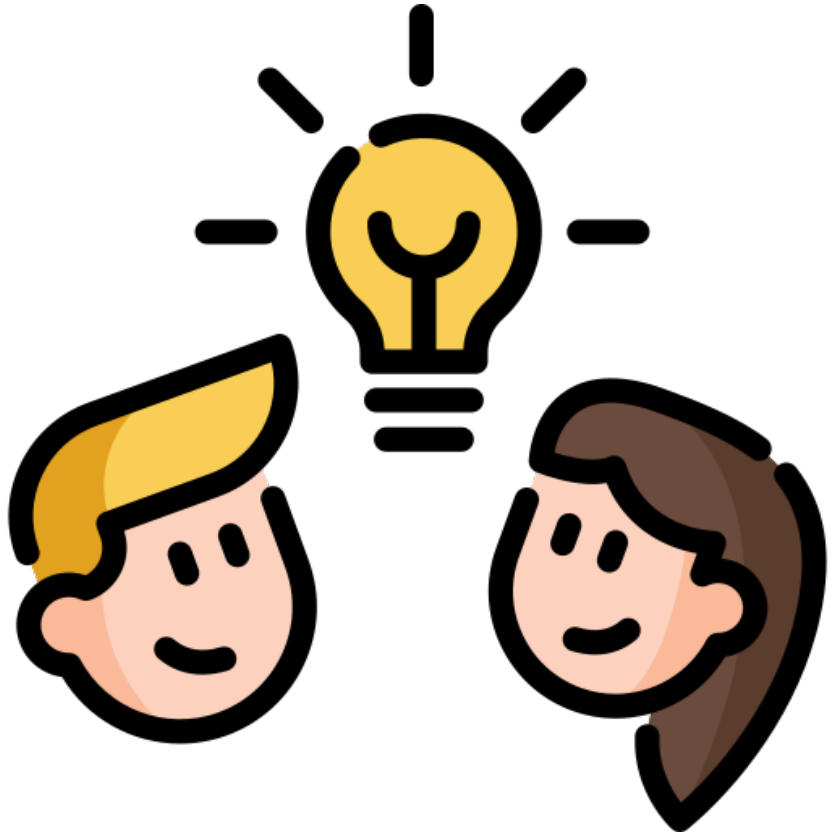


Proporciona una API para interactuar con la base de datos, lo que permite realizar operaciones como insertar, actualizar y eliminar documentos.

Fundamentos de un Driver

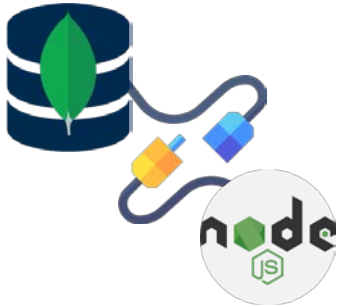
CRUD es un acrónimo que significa "Create, Read, Update, Delete" (en español: Crear, Leer, Actualizar, Eliminar). Es un término común en el desarrollo de aplicaciones y se refiere a las cuatro operaciones básicas que se pueden realizar en la mayoría de las bases de datos y aplicaciones de gestión de información.

Te invitamos a que vayas familiarizándote con éste porque será un término el cual, a partir de ahora, se volverá moneda corriente en el manejo de datos desde una aplicación Node.js.



Fundamentos de un Driver

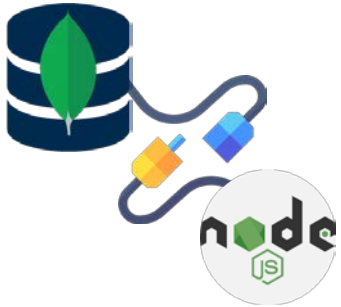
En el caso del driver oficial de MongoDB, este nos permite que las aplicaciones Node.js se comuniquen con una base de datos MongoDB de forma fácil.



Proporciona una API para interactuar con la base de datos, lo que permite realizar operaciones como insertar, actualizar, y eliminar documentos.

Fundamentos de un Driver

En Node.js existen varios drivers que nos permiten conectarnos a MongoDB. Estos se instalan como una dependencia, se referencian en nuestras aplicaciones Node.js y se utilizan mediante el lenguaje JavaScript.

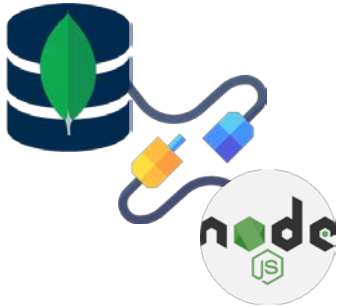


En su documentación oficial encontraremos los objetos y métodos apropiados para utilizar estos drivers correctamente.

Fundamentos de un Driver

En Node.js encontraremos dos Drivers como los más utilizados para conectarnos a bb.dd. MongoDB. **Estos son:**

- Driver oficial de MongoDB para Node.js
- Mongoose

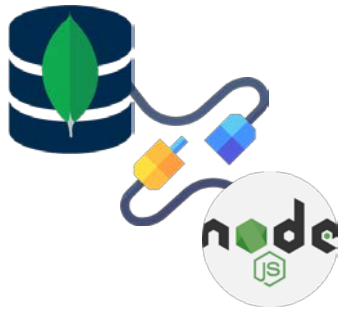


Ambos cumplen el mismo objetivo, conectarnos a MongoDB para realizar operaciones CRUD.

Fundamentos de un Driver

Driver oficial de MongoDB para Node.js:

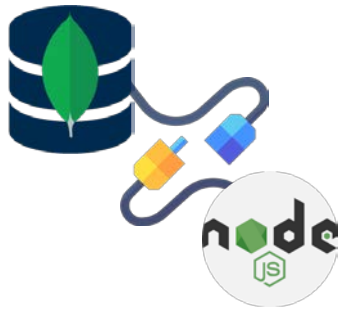
- la opción más directa y liviana para interactuar con una base de datos MongoDB desde Node.js
- Proporciona una API nativa y directa
- Posee una mayor flexibilidad y control sobre las operaciones CRUD que se realizan en la bb.dd. ya que se escribe el código
- Es más adecuado para aplicaciones que requieren una personalización completa y que no necesitan un alto nivel de abstracción sobre las operaciones de la base de datos



Fundamentos de un Driver

Mongoose:

- Proporciona una capa de abstracción sobre el driver oficial de MongoDB, lo que permite trabajar con la base de datos de manera más intuitiva y con menos código
- Incluye características que facilitan el trabajo con MongoDB, como validaciones de datos y referencias a otros documentos
- Es más adecuado para aplicaciones que requieren una alta abstracción de la base de datos y una API intuitiva y fácil de usar



Instalación de MongoDB Driver

Instalación de MongoDB Driver

Utilizaremos el controlador oficial de MongoDB para conectarnos desde nuestras aplicaciones Node.js a la bb.dd. creada en el Clúster remoto.

Este controlador se llama **MongoDB Node.js Driver** y, por supuesto, es mantenido por *MongoDB Inc*.

Instalación de MongoDB Driver

Entonces iniciaremos a continuación, en **Visual Studio Code**, un proyecto Node.js nuevo al cual llamaremos **NodeJS + MongoDB**. En este, una vez inicializado, instalamos las siguientes dependencias:

- Express JS
- dotenv

Luego, creamos una carpeta llamada **src**, dentro de ella un archivo llamado **mongodb.js** y, finalmente, el archivo de variables de entorno.

Instalación de MongoDB Driver

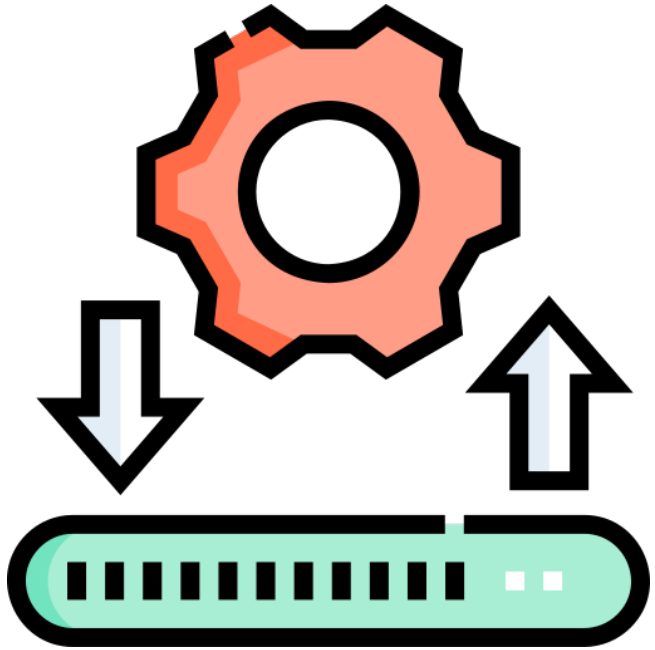
Agregaremos ahora la dependencia / Driver oficial de MongoDB, a nuestro proyecto Node.js.

MongoDB Driver oficial

```
npm install mongodb
```

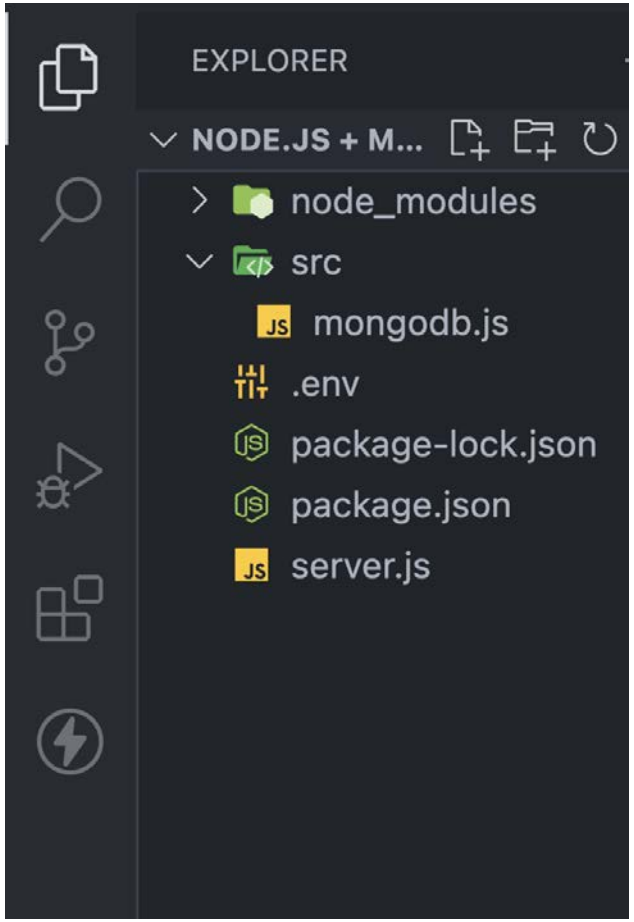
Instalación de MongoDB Driver

Con todas las dependencias y Drivers instalados, ya podemos establecer la conexión entre la aplicación Node.js y MongoDB.



Conexión de Node.js con MongoDB

Conexión de Node.js con MongoDB



Trabajaremos en el archivo de conexión con MongoDB.

Este se mantiene en la carpeta **/src** para permitirnos llevar un orden en los proyectos que con el tiempo contarán con muchos archivos.

Conexión de Node.js con MongoDB

En la estructura principal del archivo **mongodb.js** declaramos una referencia a la dependencia **dotenv**, y luego su correspondiente configuración.

mongodb.js

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

Conexión de Node.js con MongoDB

Seguido a la declaración de **dotenv**, agregamos la referencia al **Driver mongodb**. Con esto ya podremos comenzar a utilizarlo.

mongodb.js

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

const { MongoClient } = require('mongodb');
```

Conexión de Node.js con MongoDB

Es importante entender que, al depender de una aplicación externa a nuestros desarrollos Node.js, el modelo asincrónico debe estar presente dentro de la lógica de nuestra aplicación, ya que los tiempos de respuesta de un servidor Cloud escapan a nuestro control.

Uso de asincronismo

```
//función JS convencional
async function connectoToDB() {
  await client.connection()
}

Esto es sólo código de ejemplo

//Arrow function
const connectToDB = async ()=> {
  await client.connection()
}
```

Seguridad al definir una Conexión

Seguridad al definir una conexión

La seguridad en conexiones de aplicaciones backend con bases de datos es un tema importante. Si bien antes se realizaban sin prestar mucha atención a estos detalles, en la actualidad existen equipos de Seguridad Informática que analizan en detalle cuál es el mejor camino para segurizar las conexiones y el intercambio de información entre una bb.dd. y las aplicaciones clientes.

Seguridad al definir una conexión

En nuestro caso nos enfocaremos en utilizar el archivo **.env** para configurar la información básica del servidor de base de datos y así utilizar de forma segura la conexión con éste.

Seguridad al definir una conexión

La conexión que debemos establecer partirá de una URI con el protocolo de comunicación propio de MongoDB. Contamos con dos posibilidades para definir dicha conexión:

1) definir una URL, o URI, con la información que esta solicita

Seguridad al definir una conexión

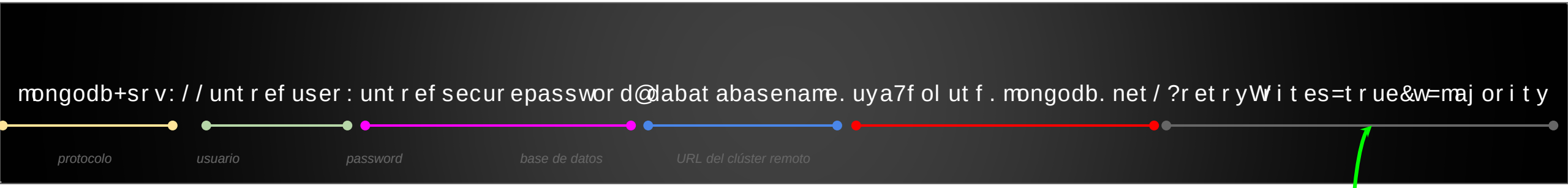
Definimos una variable de entorno y en la misma agregamos como valor la URI con la información que solicita el clúster para conectarnos remotamente a la base de datos MongoDB.

archivo .env

```
PORT=3008

MONGODB_URLSTRING=mongodb+srv://miusuariomongodb:mipasswordmongodb
@mibasededatos.elclusterremoto/
```

Seguridad al definir una conexión

Aquí tenemos la descripción de la estructura URI de conexión a MongoDB.



Los últimos parámetros suelen ser opcionales.

Seguridad al definir una conexión

2) Definir una variable de entorno por cada una de las partes que conforman la URI.

De esta forma se pueden configurar por separado y cambiar sus valores de acuerdo a la necesidad.

Seguridad al definir una conexión

Si bien es más cómodo para poder cambiar de *clúster, usuario, password, base de datos, etc*, como todas estas variables de entorno se suman a **process.env**, a veces vuelve más tedioso tener que armar luego una URI con cada variable.

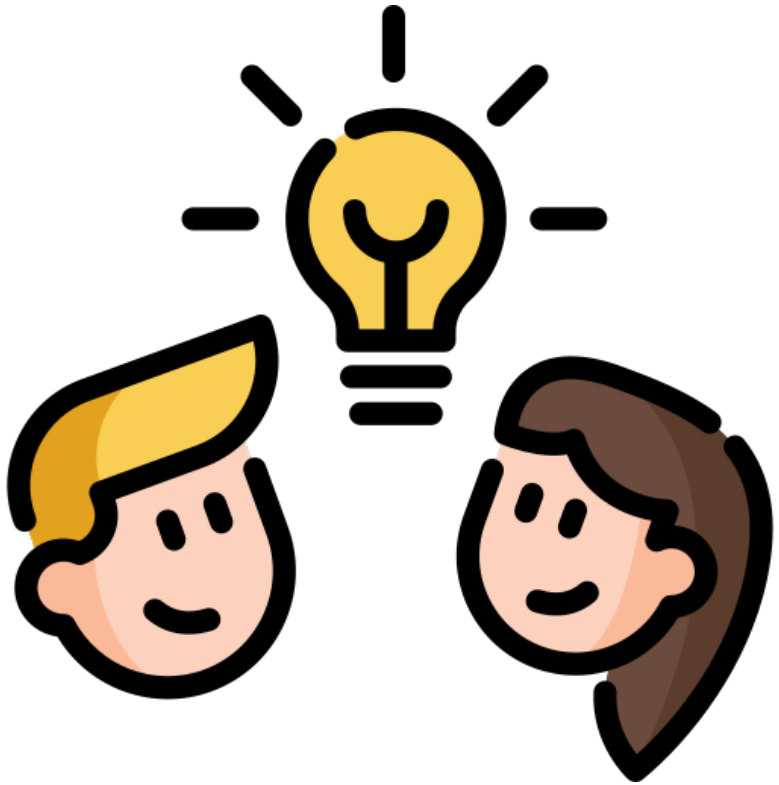
archivo .env

```
PORT=3008

MONGODB_USERNAME=untrefuser
MONGODB_PASSWORD=untr3fSecur€p@ssw0rd
MONGODB_DATABASE=dabatabasename
MONGODB_CLUSTER=uya7folutf.mongodb.net/?retryWrites=true&w=majority
```

Seguridad al definir una conexión

Si bien aquí, en un espacio de aprendizaje, somos nosotros quienes elegimos el modelo más cómodo a aplicar en la cadena de conexión, debemos tener presente que en entornos de trabajo reales será el área de seguridad informática quien nos indique cómo definir esta estructura.

Y si no existe un área dedicada, será el Arquitecto de soluciones o Team Leader quien nos guíe en cómo debemos estructurar esta cadena de conexión.



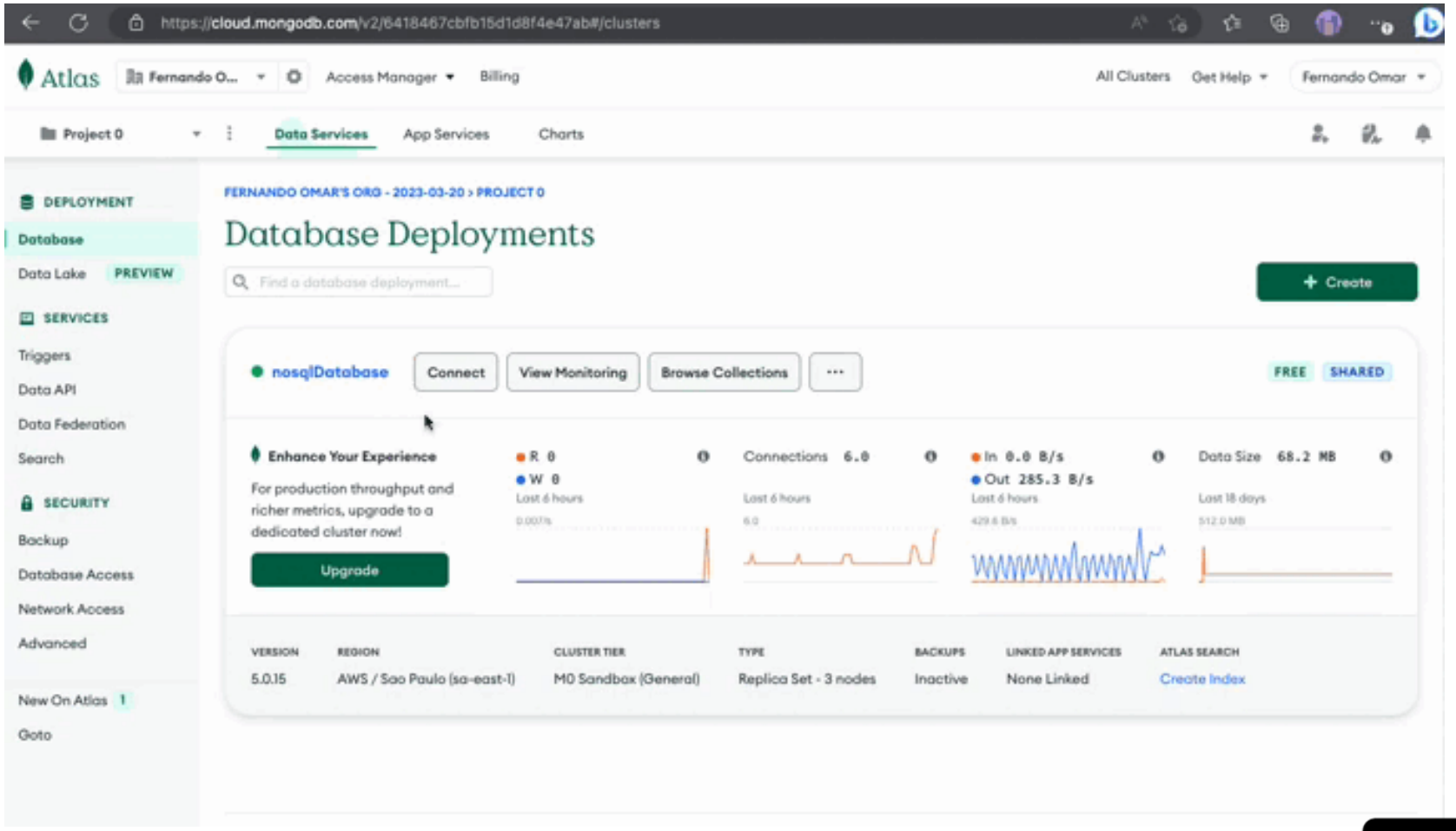
Seguridad al definir una conexión

Optaremos por definir una variable del tipo URL en el archivo **.env** y allí agregar la cadena de conexión completa de **MongoDB Clúster**.

Para obtener la **URI** de conexión, accedemos a nuestro perfil de MongoDB Atlas desde la web. Allí podremos acceder a la cadena de conexión correspondiente.

Seguridad al definir una conexión

Al acceder a MongoDB Atlas Web, encontraremos la opción de copiar la cadena de conexión MongoDB, según el lenguaje de programación que utilicemos.



Seguridad al definir una conexión

Debemos reemplazar **<password>** por el password original definido al crear la cuenta en **MongoDB Atlas**.

Con este paso realizado, nos queda crear la cadena de conexión en JS.

Acceso al motor/clúster MongoDB

Acceso al motor/clúster MongoDB

Una vez importado el Driver de MongoDB, debemos instanciar el mismo para poder utilizarlo. En dicha instancia pasaremos la URI de configuración de la conexión, a partir de una constante o de la ruta completa hacia la variable de entorno creada.

mongodb.js

```
const URI = process.env.MONGODB_URLSTRING;
const client = new MongoClient(URI);
```

Acceso al motor/clúster MongoDB

Crearemos dos funciones asincrónicas. Una para conectarnos a la bb.dd. y otra para desconectarnos.

- async connectToMongoDB()
- async disconnectFromMongoDB()

mongodb.js

```
async function connectToMongoDB() {
    try {
        await client.connect();
            console.log('Conectado a MongoDB');
        return client;
    } catch (error) {
            console.error('Error al conectar a MongoDB:', error);
        return null;
    }
}
```

Acceso al motor/clúster MongoDB

El manejo de conexión y desconexión lo controlamos mediante **try - catch - finally**.

Finalmente, exportamos ambas funciones como módulo para trabajarlas luego desde la aplicación principal de Node.js.

mongodb.js

```
async function disconnectFromMongoDB() {
      try {
        await client.close();
              console.log('Desconectado de MongoDB');
      } catch (error) {
            console.error('Error al desconectar de MongoDB:', error);
      }
}

module.exports = { connectToMongoDB, disconnectFromMongoDB };
```

Acceso al motor/clúster MongoDB

MongoDB cuenta con una serie de métodos que nos facilitarán acceder a la base de datos y sus Colecciones. Veamos a continuación cuáles son alguno de ellos:

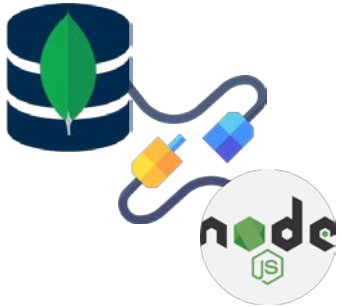
Operador	Descripción	Método
.connect()	A partir de la instancia del objeto MongoDB, este método nos permite conectarnos al motor de MongoDB.	client.connect()
.close()	A partir de la instancia del objeto MongoDB, este método cierra la conexión con el motor.	client.close()
.db()	Conectados a MongoDB, nos permite seleccionar la base de datos.	client.db(database)
.collection()	Seleccionada la bb.dd. nos permite elegir con qué colección trabajar.	db.collection(collectionName)
.find()	Busca un documento en la Colección actual, o devuelve todos los documentos.	db.collection().find(obj)
.findOne()	Busca un documento en la Colección actual, de acuerdo al objeto informado.	db.collection().findOne(obj)

Express JS y MongoDB

Express JS y MongoDB

Nuestras aplicaciones Express JS cambiarán ligeramente su estructura básica, para comenzar a soportar de forma efectiva el uso de una base de datos MongoDB.

Las funciones de retorno por cada petición, se convertirán en funciones asincrónicas para así poder controlar mejor los tiempos de respuesta del clúster de MongoDB.



Express JS y MongoDB

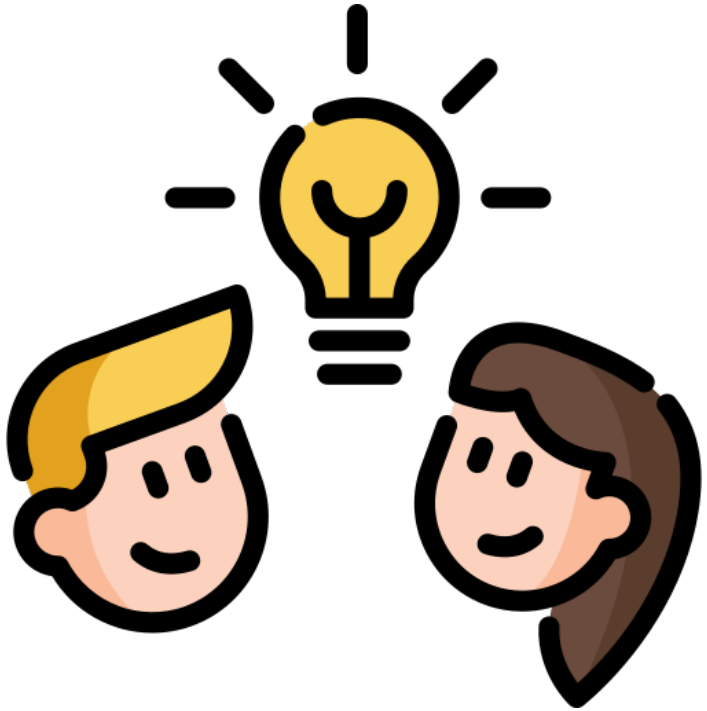
Internamente, en cada petición que se busque obtener datos de MongoDB, debemos también definir una serie de pasos para dar con ellos.

- conectarnos al motor/clúster de MongoDB
- acceder a la base de datos
- acceder a la colección
- capturar la información de respuesta
- desconectarnos del motor/clúster

Express JS y MongoDB

Es importante también controlar la conexión al clúster MongoDB, obtención de los datos, seguido de una desconexión del clúster.

De no tener en cuenta esto, las conexiones con el motor de la base de datos quedarán abiertas y esto causará una saturación de peticiones lo cual tornará desde más lenta hasta inaccesible la información de MongoDB.



Express JS y MongoDB

La estructura base de nuestro servidor web Express es igual que siempre, con el agregado de que ahora debemos importar el archivo de conexión y desconexión al clúster de MongoDB.

server.js

```
const express = require('express');
const { connectToMongoDB, disconnectFromMongoDB } = require('./src/mongodb');
const app = express();
const PORT = process.env.PORT || 3000;
```

Express JS y MongoDB

Como trabajaremos con datos en formato JSON por doquier, definiremos un Middleware global para todos los endpoint de nuestra aplicación, el cual se ocupará de establecer el **charset** en formato **utf-8**.

server.js

```
//Middleware
app.use((req, res, next) => {
    res.header("Content-Type", "application/json; charset=utf-8");
    next();
});
```

De esta forma, garantizamos el soporte de caracteres extendidos de forma global para todos los datos a visualizar.

Express JS y MongoDB

La ruta principal de nuestra aplicación de servidor será como siempre la ruta de bienvenida, enviando un mensaje básico.

server.js

```
app.get('/', (req, res) => {
    res.status(200).end('¡Bienvenido a la API de frutas!');
});
```

Acceso a una bb.dd y a una Colección MongoDB

Acceso a una bb.dd y a una Colección MongoDB

Definiremos a continuación dos rutas en nuestro servidor web.

- /frutas
- /frutas/:id

Dentro de ambas, estableceremos la estructura de conexión al clúster, base de datos, colección, para finalmente retornar los datos obtenidos.

Acceso a una bb.dd y a una Colección MongoDB

Dentro de la primera ruta, definimos la conexión al clúster, el cual retorna un objeto **client**. A través de este definimos un objeto db para conectarnos a la base de datos correspondiente y dentro de esta a la Colección **frutas**, con la cual deseamos interactuar.

server.js

```
app.get('/frutas', async (req, res) => {
    const client = await connectToMongoDB();
    const db = client.db('frutas');
    const frutas = await db.collection('frutas').find().toArray();

    await disconnectFromMongoDB();
          res.json(frutas);
});
```

Acceso a una bb.dd y a una Colección MongoDB

La misma estructura aplicamos para buscar un documento a partir del **parámetro recibido por URL**. En este caso, debemos almacenar el parámetro recibido en una variable o constante, para luego aplicar el método de filtrado correspondiente.

server.js

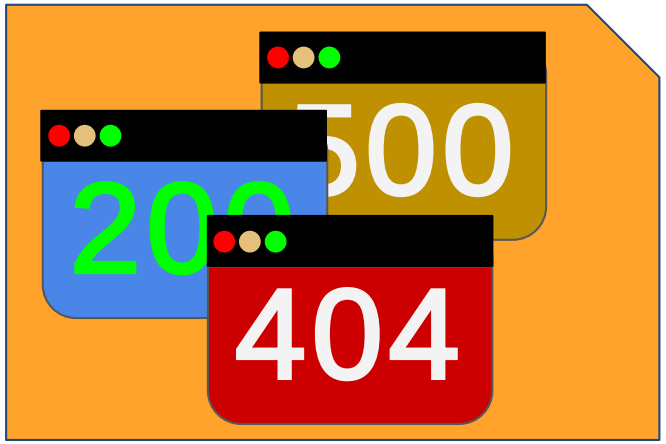
```
app.get('/frutas/:id', async (req, res) => {
    const frutaId = parseInt(req.params.id) || 0;
    const client = await connectToMongoDB();
    const db = client.db('frutas');
    const fruta = await db.collection('frutas').findOne({ id: frutaId });

    await disconnectFromMongoDB();
          res.json(fruta)
  });
```

Manejo de códigos de estado y errores

Manejo de códigos de estado y errores

En cada una de las interacciones que haremos con MongoDB, a través de nuestra aplicación Node.js, debemos tener presente conectarnos y desconectarnos de la bb.dd en el mismo proceso.

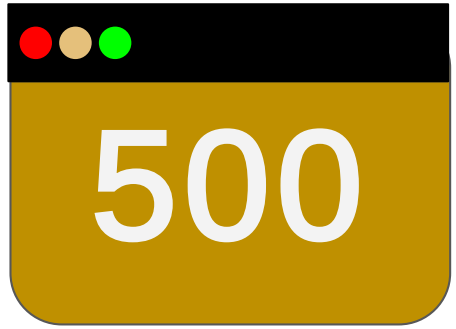
Estos procesos, sumados al de obtener datos almacenados, deben manejar diferentes códigos de estados y/o errores.



Manejo de códigos de estado y errores

El primer error que debemos controlar, es si nos es imposible conectarnos a la base de datos. En el caso de fallar la petición a dicha conexión, debemos reportar desde nuestro servidor web un **error 500**.

A partir de aquí, ninguna otra instrucción consecutiva debe ejecutarse.



Manejo de códigos de estado y errores

En nuestra lógica, al invocar la función **connectToMongoDB()**, esta devuelve un objeto de conexión, o un valor **null**.

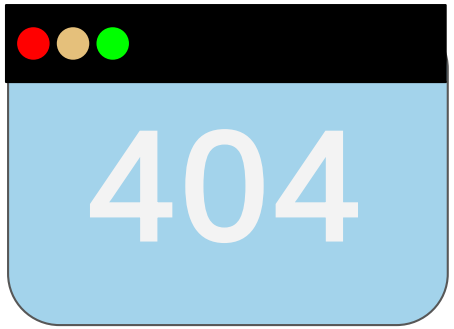
De retornar esto último, entonces retornamos un **código de estado 500** y **return**, para interrumpir el resto de la ejecución de la **petición GET**.

server.js

```
app.get('/frutas', async (req, res) => {
    const client = await connectToMongoDB();
          if (!client) {
              res.status(500).send('Error al conectarse a MongoDB');
              return;
          }
    ...
    const db = client.db('frutas');
```

Manejo de códigos de estado y errores

En el caso del **código de estado 404**, lo utilizaremos específicamente en la **petición GET** que devuelve una fruta a partir de un **ID** informado.



En el caso de no encontrar el recurso en cuestión, informaremos este error como respuesta en dicha petición.

Manejo de códigos de estado y errores

Mediante un **operador ternario** o la cláusula **if - else**, podemos controlar si disponemos de datos para retornar en base al parámetro de búsqueda recibido. En caso de no tener datos para retornar, llamamos al **código de estado 404** para notificar que no se encontró el recurso.

server.js

```
app.get('/frutas', async (req, res) => {
    const client = await connectToMongoDB();
          if (!client) {
              res.status(500).send('Error al conectarse a MongoDB');
              return;
          }
    ...
    const db = client.db('frutas');
```

Prácticas

Prácticas

Define nuevas rutas como endpoint para retornar diferentes tipos de datos:

1. /frutas/nombre/:nombre
2. /frutas/precio/:precio

En la primera de las rutas, deberás **buscar y retornar todas aquellas frutas que contengan el nombre o parte** del nombre informado como parámetro.

En el segundo endpoint definido, deberás **buscar y retornar todas aquellas frutas que tengan el mismo precio informado o un precio superior** a este.

En estos endpoints, deberás utilizar el método **.find().toArray()**, para que devuelva más de un resultado.

También **deberás integrar Expresiones Regulares**. JavaScript las trabaja de forma nativa con el objeto **RegExp()**. Te compartimos documentación aquí.

Muchas gracias.

Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

primero
la gente