

Desarrollo Backend

Bienvenid@s

Node.js y MongoDB

(Desarrollar un CRUD completo)

Clase 17



Pon a grabar la clase



Temario

- El concepto de CRUD
- Cómo desarrollar un CRUD con MongoDB y

Express

- Lectura de datos
 - Grabar un nuevo recurso
 - Modificar un recurso existente
 - Eliminar un recurso
- Testear nuestra API Restful



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

El concepto de CRUD

El concepto de CRUD

El término **CRUD** es un acrónimo que se utiliza en el mundo de la programación para describir las operaciones básicas que se pueden realizar sobre una base de datos.

CRUD significa "**C**reate, **R**ead, **U**psdate, **D**elete" (*Crear, Leer, Actualizar y Borrar, respectivamente*).



El concepto de CRUD

Estas cuatro operaciones básicas describen las acciones que se pueden llevar a cabo sobre la información almacenada en una base de datos.

CREATE: Añadir recursos en una bb.dd.

READ: Consulta recursos en una bb.dd.

UPDATE: Modifica recursos en una bb.dd.

DELETE: Elimina recursos de una bb.dd.



El concepto de CRUD

El término **CRUD** es también conocido en español como **ABM (Alta, Baja y Modificación)** de datos en una bb.dd.

!: Si bien siempre nos referenciamos a “*datos*” cuando hablamos de una bb.dd., también es común escuchar hablar en referencia a operaciones CRUD, como “*recursos*” en un servidor.



El concepto de CRUD

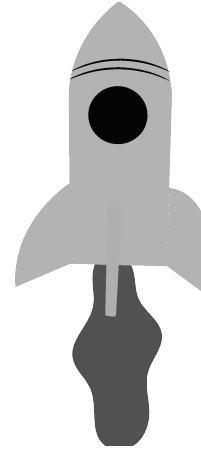
Si bien CRUD permite trabajar con bases de datos, también es aplicable a crear recursos en un servidor como, por ejemplo, subir un archivo a un servidor web, eliminar un archivo en un servidor web, o editar el contenido de un archivo en un servidor web.

En estos casos mencionados es cuando hablamos de “recursos” y no de datos específicos.

El concepto de CRUD

El concepto de CRUD es importante porque **proporciona una forma estructurada y consistente** la posibilidad de **interactuar con datos almacenados en una bb.dd.**

Esto hace que sea más fácil para toda aquella persona que cumpla el rol de software developer, poder diseñar y desarrollar aplicaciones que utilizan una base de datos, ya que pueden seguir este conjunto de operaciones básicas para realizar las acciones necesarias sobre los datos.



El concepto de CRUD

En el contexto de **MongoDB + Node.js + Express**, se puede utilizar la misma estructura de aplicaciones que venimos trabajando hasta el momento, para agregar, modificar, y eliminar recursos de la base de datos MongoDB.

Veamos entonces qué herramientas debemos integrar de Express para llevar a cabo un CRUD completo contra MongoDB de acuerdo a la aplicación base que estamos construyendo en nuestros últimos encuentros.



Cómo desarrollar un CRUD con MongoDB y Express

Cómo desarrollar un CRUD con MongoDB y Express

En el proyecto de **Node.js + Express JS + MongoDB** que **venimos evolucionando**, tenemos incluido hasta el momento la petición **GET** de todos los documentos de la bb.dd. **frutas**, y algunas peticiones GET puntuales, que nos permiten aplicar filtros sobre la colección en cuestión.

Para poder concretar el resto de las operaciones CRUD, traeremos a la Arena nuevamente a los siguientes métodos:



Cómo desarrollar un CRUD con MongoDB y Express

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:

Método	Descripción
.post()	El método .post() que forma parte de Express JS nos permite crear nuevos recursos en el servidor. La información que le enviamos a éste, viajará a través del cuerpo 'body' de la petición que realizaremos.
.put()	El método .put() también permite crear un nuevo recurso en el servidor, o modificar uno existente. Comúnmente es más utilizado para esto último. Para modificar un recurso de servidor, debemos enviar algún identificador del mismo por parámetro de la URL, y en el cuerpo de la petición, los datos a modificar.
.delete()	El método .delete() se ocupa de eliminar un recurso del servidor. No utiliza un cuerpo de petición pero sí espera un parámetro por URL para identificar el recurso previo a eliminarlo.

Cómo desarrollar un CRUD con MongoDB y Express

La intención de este proyecto es continuar evolucionando el trabajo que realizamos en nuestra última clase.

Al proyecto Express JS con conexión al clúster MongoDB, y los métodos GET que nos permiten obtener una colección de frutas de acuerdo a los diferentes parámetros, le sumaremos la interacción realizada mediante CRUD, interviniendo sobre la misma bb.dd. MongoDB.



Lectura de datos

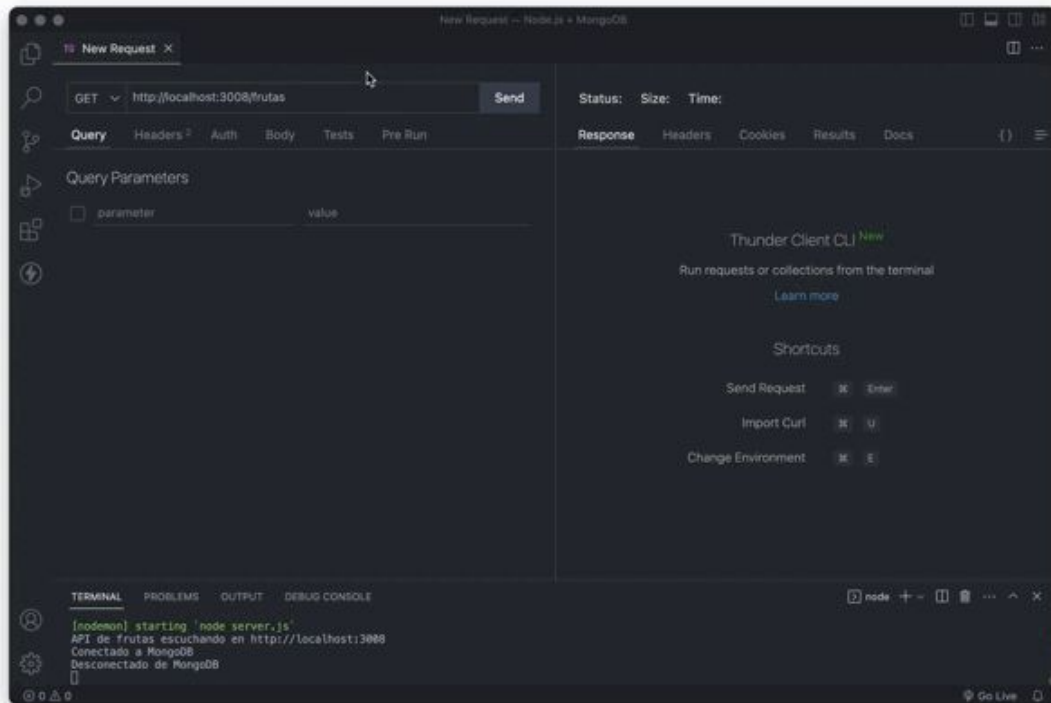
Lectura de datos

La lectura de datos ya la tenemos resuelta desde nuestra clase anterior.

Gracias al método **app.get()** podemos acceder a la información de la Colección frutas de MongoDB para realizar una lectura completa de los documentos almacenados, aplicar un filtro por **:id**, y también por **:nombre** y por **:precio**.

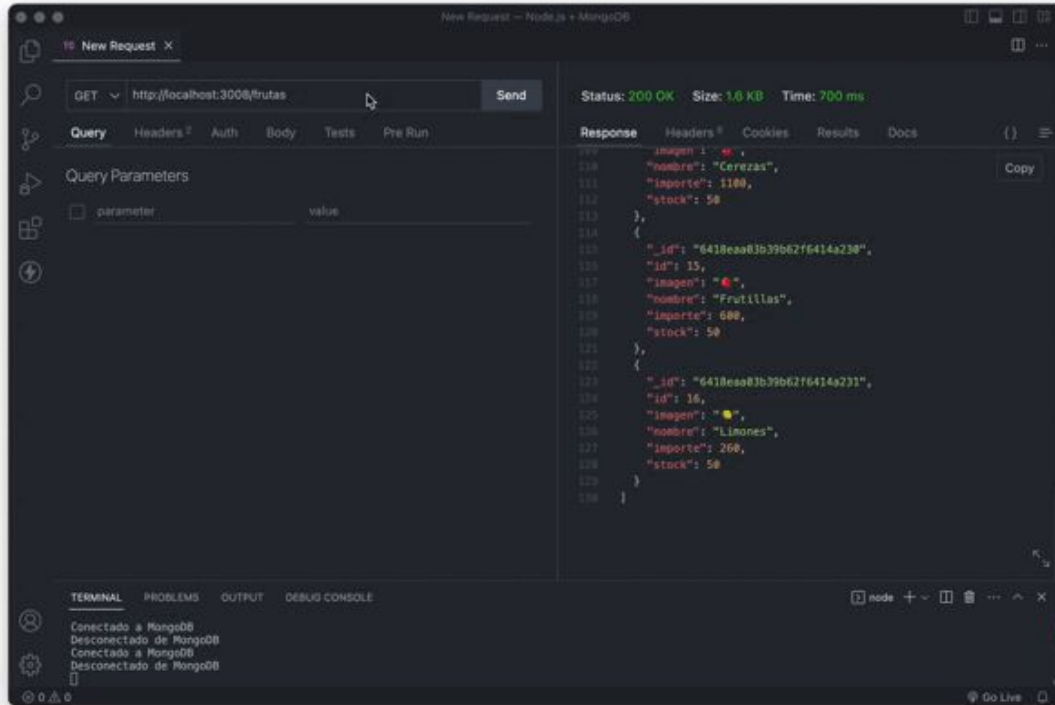


Lectura de datos



Petición **GET** general,
que nos retorna todos los
documentos
almacenados en la
Colección frutas.

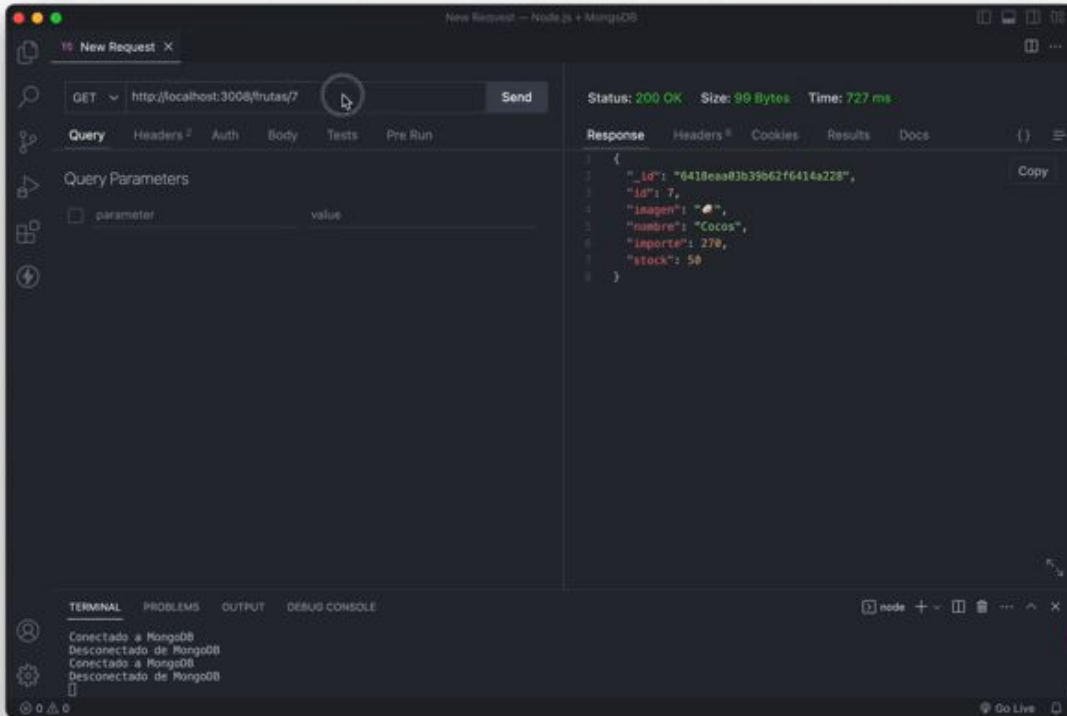
Lectura de datos



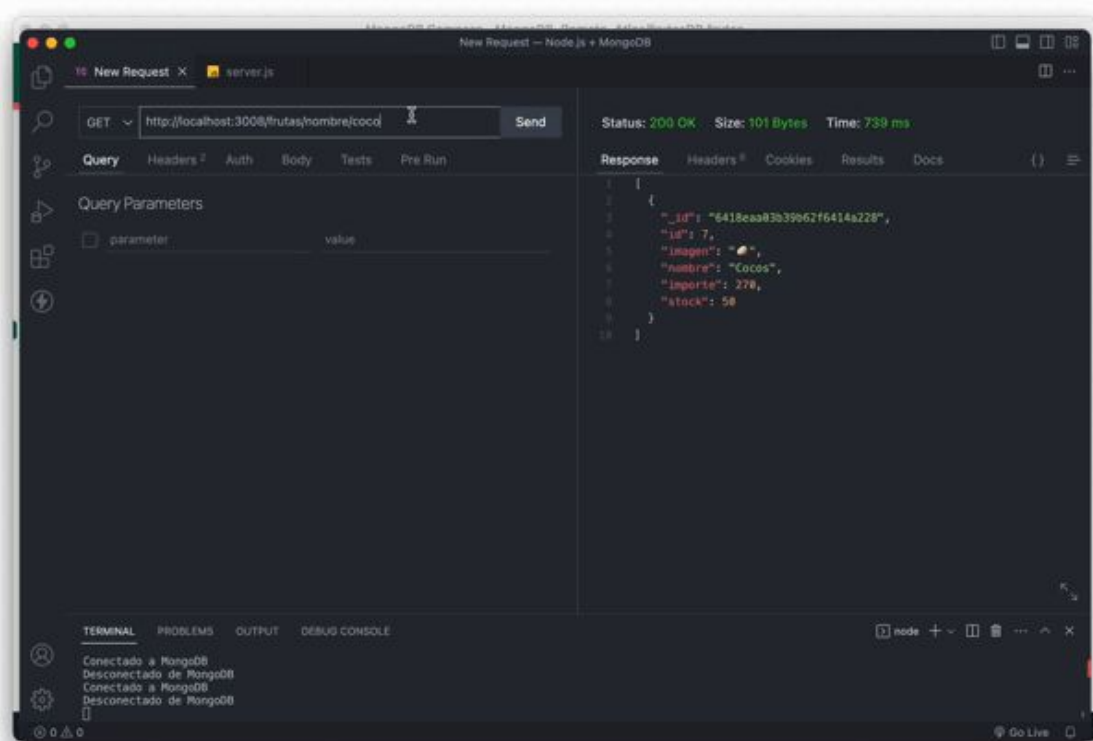
Petición GET buscando un documento por su atributo **:id**.

Lectura de datos

Petición GET buscando un documento por su atributo **:nombre**, o parte del mismo.



Lectura de datos



Petición GET buscando un documento por su atributo **:importe**, igual o superior al valor indicado.

Crear un recurso (*nuevo documento*)

Crear un nuevo recurso

Crearemos un nuevo recurso en el servidor.

Para ello, recurriremos al método **.post()** integrado en Express JS.

En esta interacción debemos tener presente algunos factores específicos.



Crear un nuevo recurso

En primer lugar, habíamos visto que MongoDB maneja su propio identificador, denominado **objectId()**.

Este se crea de forma automática cuando agregamos un nuevo documento en dicha **bb.dd** y seguirá comportándose de igual forma cuando interactuemos desde Express JS.


```
objectId()

{
  "_id": "6418eaa03b39b62f6414a22f",
  "id": 14,
  "imagen": "🍒",
  "nombre": "Cerezas",
  "importe": 1100,
  "stock": 50
}
```

Crear un nuevo recurso

Otra cuestión a tener en cuenta, aunque la pasaremos por alto, es que nuestra colección de datos (*importada desde un archivo JSON*) tiene su propio identificador, numérico, y consecutivo.

En nuestras pruebas de grabación con el método POST, asumimos que estamos controlando correctamente dicho identificador, para no sumarle otra capa más de complejidad a nuestro código.



```
objectId()

{
  "_id": "6418eaa03b39b62f6414a22f",
  "id": 14,
  "imagen": "🍒",
  "nombre": "Cerezas",
  "importe": 1100,
  "stock": 50
}
```


Crear un nuevo recurso

También debemos diseñar la lógica correcta para poder grabar un nuevo recurso en la base de datos en cuestión.

Para ello, lo primero que debemos realizar es verificar que la petición POST que será enviada a nuestro servidor viaja con los datos correspondientes en el cuerpo (*body*). Si esta información no llega, debemos evitar seguir con la grabación del documento.



Crear un nuevo recurso

También debemos tener presente que, en el manejo de errores, tenemos que informar correctamente el código de estado del servidor de acuerdo al problema detectado.

En este caso anterior, la petición de respuesta negativa del servidor, será a través del código de estado 400.

```
petición POST

const nuevaFruta = req.body;
if (nuevaFruta === undefined) {
  res.status(400).send('Error en el formato de datos a crear.');
```

Crear un nuevo recurso

Pasada la validación de los datos recibidos en el cuerpo de la petición, debemos intentar conectarnos con el servidor MongoDB.

Aprovechamos que tenemos una función que nos retorna el éxito o error en la conexión en sí, por lo tanto es la próxima validación que haremos.



```
const client = await connectToMongoDB();
if (!client) {
  res.status(500).send('Error al conectarse a MongoDB');
}
```

Crear un nuevo recurso

Ante la falla en la conexión con el servidor MongoDB, retornaremos un **código de estado 500**, indicando que dicho intento de conexión falló.



petición POST

```
const client = await connectToMongoDB();  
if (!client) {  
  res.status(500).send('Error al conectarse a MongoDB');  
}
```

Crear un nuevo recurso

Pasada ambas validaciones, nos ocupamos de conectarnos a la colección, declarando la misma en una constante homónima.

A partir de allí, invocamos al método **.insertOne()**, propio de la dependencia MongoDB.

A dicho método le pasamos como parámetro la constante **nuevaFruta**.

petición POST

```
const collection = client.db('frutasDB').collection('frutas');  
collection.insertOne(nuevaFruta)
```

Crear un nuevo recurso

El método **insertOne()** retorna una promesa.

Aprovechando la misma, estructuramos a través del método **then()** el control de la operación realizada y el envío correspondiente del código de estado (201).

catch() nos ayuda a capturar cualquier error que acontezca.

finally() nos permite cerrar la conexión más allá del estado exitoso o fallido de dicho método.

petición POST

```
collection.insertOne(nuevaFruta)
  .then(() => {
    console.log('Nueva fruta creada:');
    res.status(201).send(nuevaFruta);
  })
  .catch(error => {
    console.error(error);
  })
  .finally(() => {
    client.close();
  });
```

Crear un nuevo recurso

Aquí tenemos el código completo, correspondiente al método **.post()**.

Como podemos apreciar, el método **post** lo trabajamos como una función asíncronica, combinando la misma con el retorno de una promesa del método **insertOne()**, de **mongoDB**.

```
petición POST

app.post('/frutas', async (req, res) => {
  const nuevaFruta = req.body;
  if (nuevaFruta === undefined) {
    res.status(400).send('Error en el formato de datos a crear.');
```

```
  }

  const client = await connectToMongoDB();
  if (!client) {
    res.status(500).send('Error al conectarse a MongoDB');
  }

  const collection = client.db('frutasDB').collection('frutas');
  collection.insertOne(nuevaFruta)
    .then(() => {
      console.log('Nueva fruta creada:');
      res.status(201).send(nuevaFruta);
    })
    .catch(error => {
      console.error(error);
    })
    .finally(() => {
      client.close();
    });
});
```

Modificar un recurso

Modificar un recurso

Ahora nos toca modificar un recurso existente en el servidor. Para ello, usaremos el método **.put()** integrado en Express JS.

En el proceso de actualización, recurriremos al método **updateOne()** el cual retornará una promesa que nos permitirá controlar el proceso de forma efectiva.



Modificar un recurso

La lógica de control aplicada en el método post para con los datos que recibimos mediante body como también con la conexión al servidor, se mantienen tal cual lo usamos en este último método.

```
petición PUT

const id = req.params.id;
const nuevosDatos = req.body;

if (!nuevosDatos) {
  res.status(400).send('Error en el formato de datos recibido.');
```



```
const client = await connectToMongoDB();
if (!client) {
  res.status(500).send('Error al conectarse a MongoDB');
```

Modificar un recurso

En el caso del método **put()**, debemos informarle dos parámetros. El primero de ellos es el identificador del documento que modificaremos. Utilizaremos el id propio de nuestra colección, para no tener que lidiar con una estructura.

petición PUT

```
const collection = client.db('frutasDB').collection('frutas');  
collection.updateOne({ id: parseInt(id) }, { $set: nuevosDatos })
```

Modificar un recurso

El segundo de los datos a enviar, corresponde a la información recibida en el cuerpo de la petición. Al igual que con el método POST, no necesitamos informar el **objectId()** de la misma.

petición PUT

```
const collection = client.db('frutasDB').collection('frutas');  
collection.updateOne({ id: parseInt(id) }, { $set: nuevosDatos })
```

Modificar un recurso

El símbolo **\$set** es un operador de actualización utilizado en MongoDB para actualizar un campo específico en un documento existente. Cuando se usa con el método **updateOne()** permite actualizar uno o varios campos de un documento sin tener que reemplazarlo todo.

petición PUT

```
collection.updateOne({ id: parseInt(id) }, { $set: nuevosDatos })
```

Modificar un recurso

Si nuestro método **.put()** se ocupará de modificar un campo específico, y supiéramos de antemano cuál será dicho campo, podemos especificar el mismo con su valor correspondiente, evitando referenciar todo el array completo, tal como haremos en este ejemplo.



petición PUT

```
collection.updateOne(  
  { "id": 14 },  
  { $set: { "precio": 1070 } }  
)
```

Modificar un recurso

Aquí tenemos el código completo de este método.

Como podemos apreciar, el código de estado de una **modificación efectiva será el 200**. Cualquier **otro error será controlado por el código de estado 500**.

```
petición PUT

app.put('/frutas/:id', async (req, res) => {
  const id = req.params.id;
  const nuevosDatos = req.body;

  if (!nuevosDatos) {
    res.status(400).send('Error en el formato de datos recibido.');
```

```
  }

  const client = await connectToMongoDB();
  if (!client) {
    res.status(500).send('Error al conectarse a MongoDB');
  }

  const collection = client.db('frutasDB').collection('frutas');
  collection.updateOne({ id: parseInt(id) }, { $set: nuevosDatos })
    .then(() => {
      console.log('Fruta modificada:');
      res.status(200).send(nuevosDatos);
    })
    .catch((error) => {
      res.status(500).json({descripcion: 'Error al modificar la fruta'});
    })
    .finally(() => {
      client.close();
    });
});
```

Eliminar un recurso

Eliminar un recurso

Por último, nos queda eliminar un recurso existente en el servidor. Para ello, usaremos el método **.delete()** integrado también en el framework Express JS.

En este proceso identificamos el recurso por su id, para luego poder eliminarlo de la colección en cuestión.



Eliminar un recurso

La ruta de eliminación de un recurso debe recibir como parámetro del tipo URL Params, el id del producto que se desea eliminar.

Validamos entonces que el dato recibido por parámetro sea correcto, sino respondemos la petición con el código de error 400.

```
Petición DELETE

const id = req.params.id;
if (!req.params.id) {
  return res.status(400).send('El formato de datos es erróneo o inválido.');
```



```
const client = await connectToMongoDB();
if (!client) {
  return res.status(500).send('Error al conectarse a MongoDB');
```

Eliminar un recurso

Luego de pasar la primera validación, seguimos con la validación de poder conectarnos correctamente a la bb.dd. MongoDB.

Pasada ambas validaciones, ya podemos ocuparnos de eliminar el recurso en cuestión.

```
Petición DELETE

const id = req.params.id;
if (!req.params.id) {
  return res.status(400).send('El formato de datos es erróneo o inválido.');
```



```
const client = await connectToMongoDB();
if (!client) {
  return res.status(500).send('Error al conectarse a MongoDB');
```

Eliminar un recurso

Conectados a la bb.dd. ya iniciamos el control del proceso de eliminación mediante el retorno de la promesa del método **client.connect()**. Dentro del primer método de control **then()** invocamos al método **deleteOne()** integrado en el **objeto Collection**.

Petición DELETE

```
client.connect()
  .then(() => {
    const collection = client.db('frutasDB').collection('frutas');
    return collection.deleteOne({ id: parseInt(id) });
  })
```

Eliminar un recurso

El método **deleteOne()** espera un objeto como parámetro. Allí debemos indicarle la propiedad **id** del documento junto con el valor que recibimos como parámetro. De forma implícita, **deleteOne()** retorna un valor como resultado. Este dato en cuestión, lo pasaremos como parámetro al siguiente método de control **then()**.

Petición DELETE

```
client.connect()
  .then(() => {
    const collection = client.db('frutasDB').collection('frutas');
    return collection.deleteOne({ id: parseInt(id) });
  })
```

Eliminar un recurso

En el segundo método de control **then()**, pasamos un parámetro que asume el retorno implícito que proviene del método anterior; en nuestro ejemplo lo llamamos **resultado**.

Este parámetro es un objeto, y dentro de este objeto encontraremos una propiedad llamada **deletedCount**.

```
Petición DELETE

.then((resultado) => {
  if (resultado.deletedCount === 0) {
    res.status(404).send('No se encontró ninguna fruta con el ID:', id);
  } else {
    console.log('Fruta eliminada.');
    res.status(204).send();
  }
})
```

Eliminar un recurso

La propiedad **deletedCount** nos provee el valor numérico de el o los documentos eliminados por el método **deleteOne()**. Si vuelve **0** (cero), es porque no encontró el **id** informado como parámetro.

```
Petición DELETE

.then((resultado) => {
  if (resultado.deletedCount === 0) {
    res.status(404).send('No se encontró ninguna fruta con el ID:', id);
  } else {
    console.log('Fruta eliminada.');
    res.status(204).send();
  }
})
```

Eliminar un recurso

En esta última instancia, retornaremos un **código de estado 204**, apropiado para indicar que el recurso ya no existe en el servidor.

```
Petición DELETE

.then((resultado) => {
  if (resultado.deletedCount === 0) {
    res.status(404).send('No se encontró ninguna fruta con el ID:', id);
  } else {
    console.log('Fruta eliminada.');
    res.status(204).send();
  }
})
```


Eliminar un recurso

Por último, controlamos cualquier error general no contemplado, a través del método **catch(error)**, y con el método **finally()** nos ocupamos de cerrar la conexión, hayamos realizado la eliminación del recurso de forma correcta, o no.



```
.catch((error) => {  
  console.error(error);  
  res.status(500).send('Se produjo un error al intentar eliminar la fruta.');
```

```
  })  
  .finally(() => {  
    client.close();  
  });
```

Eliminar un recurso

Aquí tenemos el código completo del método delete, con todos los controles requeridos y la ejecución del método

deleteOne() previendo cada paso del proceso con el control de los métodos **then()**, propios del retorno de una promesa.

```
Petición DELETE

app.delete('/frutas/:id', async (req, res) => {
  const id = req.params.id;
  if (!req.params.id) {
    return res.status(400).send('El formato de datos es erróneo o inválido.');
```

```
  }

  const client = await connectToMongoDB();
  if (!client) {
    return res.status(500).send('Error al conectarse a MongoDB');
```

```
  }

  client.connect()
    .then(() => {
      const collection = client.db('frutasDB').collection('frutas');
      return collection.deleteOne({ id: parseInt(id) });
    })
    .then((resultado) => {
      if (resultado.deletedCount === 0) {
        res.status(404).send('No se encontró fruta con el ID proporcionado:', id);
      } else {
        console.log('Fruta eliminada.');
```

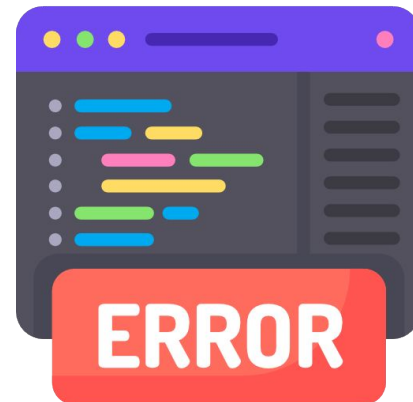
```
        res.status(204).send();
      }
    })
    .catch((error) => {
      console.error(error);
      res.status(500).send('Se produjo un error al intentar eliminar la fruta.');
```

```
    })
    .finally(() => {
      client.close();
    });
});
```

Testear nuestra API Restful

Testear nuestra API Restful

Seguramente en esta etapa de pruebas no encontremos errores significativos, pero cuando nuestra API REST dentro de una empresa de tecnología, sea encarada por los equipos de testing, allí seguramente encontraremos errores que pasamos por alto en el desarrollo inicial.



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Testear nuestra API Restful

Los equipos de testing están preparados
justamente para eso:

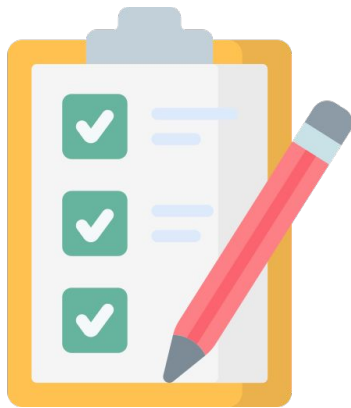
“*encontrar errores*” y ayudarnos a solucionar los
mismos de cara a que nuestro producto se
implemente en un ambiente real, de la forma más
pulida posible.



Testear nuestra API Restful

Vamos entonces a elaborar el listado básico de pruebas que debemos realizar:

- la URL o ruta principal
- la URL general para visualizar todos los productos
- la URL que nos retorna un producto por su ID, por su nombre, y por un precio aproximado
- la URL que nos permite dar de alta un recurso
- la URL que nos permite modificar un recurso existente
- la URL que nos permite eliminar un recurso



¡Parece poco, pero es mucho! 🤖

Espacio de trabajo

Espacio de trabajo

De acuerdo al ejemplo presentado por la profe y al modelo de datos que estamos utilizando en MongoDB, aportemos entre todos cuáles serán los parámetros posibles que debemos utilizar para probar cada uno de los ENDPOINT que representaremos en la siguiente diapositiva.

En un modo brainstorming, debemos indicar qué tipo de valor válido o efectivo podemos usar para testear cada endpoint y hasta incluso qué valores no válidos podemos aplicar también.

De esta forma lograremos no solo probar el camino feliz de la API REST, sino también verificar cómo esta se comporta ante valores que no son esperados recibir como parámetros del lado del BACKEND.

Tiempo estimado: 20 minutos. 

Espacio de trabajo

URL ¹	Descripción	Método
<code>http://localhost:3008/</code>	La URL o ruta principal	GET
<code>http://localhost:3008/frutas</code>	La URL general para visualizar todos los productos	GET
<code>http://localhost:3008/frutas/:id</code>	La URL que nos retorna un producto por su ID	GET
<code>http://localhost:3008/frutas/nombre/:nombre</code>	La URL que nos retorna un producto por su nombre	GET
<code>http://localhost:3008/frutas/importe/:precio</code>	La URL que nos retorna un producto por su precio aproximado	GET
<code>http://localhost:3008/frutas/</code>	La URL que nos permite dar de alta un recurso	POST
<code>http://localhost:3008/frutas/:id</code>	La URL que nos permite modificar un recurso existente	PUT
<code>http://localhost:3008/frutas/:id</code>	La URL que nos permite eliminar un recurso	DELETE

¹ En este caso, la ruta base de la API REST puede variar en nuestro entorno de pruebas, si es que el puerto definido por nosotras difiere al de los ejemplos representados en esta diapositiva.

```
const questions = [ 'dudas', 'consultas', '🤔' ]
```



```
> node gracias.js
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO