REST API

En uno de los ejercicios vimos cómo traer información de una API externa. Hicimos una consulta HTTP y nos devolvió un objeto JSON con un montón de información de productos que pudimos utilizar en nuestro proyecto.

La idea ahora es poder armar nuestra propia API para que desde nuestra web, hagamos consultas.

¿Qué es una API?

¿Qué es una REST API?

API: Conjunto de herramientas que nos permiten crear software y conectar unos con otros.

REST

Representational State Transfer
- El cliente da la información sobre lo que tiene que hacer
- CRUD

CLIENTE
- Necesito agregar un usuario
- Necesito ver un usuario
- Necesito traer posteos

METODOS HTTP/VERBOS
- GET
- POST
- PATCH
- DELETE

SERVIDOR
- Recibe la consulta del cliente y hace algo en consecuencia.

Operaciones Típicas

CRUD

Create
- POST /producto

Ej: Crear un nuevo producto

Read
- GET /producto
- GET /producto/:id (placeholder a ser reemplazado)

Update
- PATCH /producto/:id

Delete
- DELETE /producto/:id

Qué se manda a través del HTTP

Texto

Header: key value pairs que permiten agregar información al request
Accept
Authorization
Request {}

La data que mandamos, la mandamos como JSON en texto.
Vamos a pasear el texto y operar el JSON
El servidor da una response

Estructura

Project

- src
  - db
    - mongoose.js
  - index.js

Mongoose

Está relacionada 100% con MongoDb. Ya sabemos las operaciones bases con Mongo… pero hay otro tipo de validaciones necesarias

¿Cómo sé el schema?
¿Cómo seteo validaciones?
¿Cómo sé quién es el usuario que está ejecutando una acción?

https://mongoosejs.com/

Modelos

Mongoose nos va a permitir modelar nuestra base de datos. ¿Qué es un modelo? Una abstracción de algo particular a ser almacenado, con todas sus propiedades:

Usuarios

- Nombre
- Apellido
- Dni

Productos

- Nombre

- Precio
- SKU
- Marca

Etc, etc.

Mongoose nos va a llenar de métodos y propiedades para poder hacer más fácil el modelado de nuestra DB.

Utilizando mongoose

1. Es un paquete de 3ros por lo cuál vamos a usar el comando **npm i mongoose**
2. Requerimos el paquete generamos la conexión. Vamos a ver que es muy parecida a lo que hicimos con Mongo, a excepción de algunas cosas:

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

mongoose.connect('mongodb://127.0.0.1:27017/ecommerce', {
  useNewUrlParser: true,
  useCreateIndex: true
});
```

Como vemos, una de las primeras diferencias es que ahora estamos conectándonos a la dirección de la base, pero también estamos utilizando al final de la dirección, el nombre de la base de datos.

Luego, dos propiedades necesarias para utilizar con mongoose.

Modelando objetos

Para modelar un objeto vamos a utilizar el método .model() de mongoose, vamos a crear nuestra collection. Vamos a poder definir las distintas propiedades así como especificar el tipo de dato de cada propiedad. Esto nos trae una capa de seguridad para evitar guardar tipos de datos erróneos en la base de datos.

```
const Product = mongoose.model('Product', {
  name: {
    type: String
  },
  price: {
    type: Number
  }
});
```

El método .model() recibe dos parámetros

1. El nombre de la colección a crear
2. La configuración de sus propiedades. A estas les podemos especificar:
    1. Tipo
    2. Validaciones

Y para guardar un nuevo dato, creamos una instancia de nuestro modelo, y utilizamos el método .save(). Este método funciona con promesas.

1. Creo una instancia de nuestro modelo

```
const product = new Product({
  name: 'Guitarra Fender',
  price: 29000
});
```

2. Ejecuto el método .save()
3. Cuando esté guardado *.then()*, ejecuto la función de callback y ejecuto lo que necesite. Y también generamos un .catch() para saber si existen o no errores

```
product.save().then(() => {
  console.log(product);
}).catch((error) => {
  console.log(error);
})
```

Validación y Sanitización de datos

Por ahora vimos una simple validación que es la del tipo de dato a guardar en la base de datos. Esta validación nos permite mantener consistencia en los datos que ingresamos.

Built in Validators

Existe en Mongoose, lo que se llaman built-in validators. Del a misma manera que tenemos el type, esto nos permite otro tipo de validaciones.

Required

```
price: {
```

```
    type: Number,
    required: true
  }
```

Custom Validations

Existe una función llamada validate que recibe como parámetro el valor del dato que se está intentando guardar:

```
validate() {}
```

Esto nos va a permitir crear validaciones propias, como por ejemplo evitar una longitud específica.

Lo que vamos a poder hacer, es tirar un error de manera manual, con JS mediante la palabra reservada throw new Error();

```
validate(value) {
      if(value.length > 20) {
        throw new Error('El nombre tiene que tener menos de 20 caracteres')
      }
    }
```

Todo unido:

```
const Product = mongoose.model('Product', {
  name: {
    type: String,
    validate(value) {
      if(value.length > 20) {
        throw new Error('El nombre tiene que tener menos de 20 caracteres')
      }
    }
  },
  price: {
    type: Number,
    required: true,
  }
});
```

Creando los primeros endpoints

Verbo POST

Vamos a crear un endpoint que reciba una consulta HTTP **de tipo POST.** Desde el cliente vamos a mandar la información (en un objeto JSON) que queremos que persista en la DB.

Si queremos crear un producto, vamos a tener un formulario del lado del cliente que tenga un input por cada propiedad que definimos en nuestro modelo y al hacer click en el botón **“Agregar Producto”** vamos a mandar la información de nuestro formulario a la DB.

Para crear el primer endpoint, lo que vamos a hacer es utilizar el método que existe en nuestro **app** que va a ser similar al verbo que queramos usar, en este caso, POST.

```
app.post();
```

Como pasaba cuando hacíamos el app.get(), este método va a recibir dos parámetros. La ruta y el callback.

```
app.post('/products', (req, res) => {
  res.send('Funciona el POST');
});
```

Con esto acabamos de crear nuestro primer endpoint que va a recibir información del lado del cliente y va a agregarlo en la base de datos. La pregunta es… ¿Cómo recibo información del lado del cliente? Y la respuesta es **a través de un form.** Pero antes de crear el formulario, como estamos del lado del BackEnd y no necesariamente tenemos que esperar a que una persona del FrontEnd realice ese formulario, lo que vamos a usar es **Postman.**

Postman

Es un programa que nos va a permitir hacer consultas HTTP e interactuar con nuestra API sin necesidad del Front End.

Recibiendo información del cliente

Ya sabemos que vamos a recibir información del cliente y que vamos a estar usando Postman para enviar los datos, pero antes de recibir esos datos del lado del cliente, lo que vamos a necesitar es hacer que express reciba esos datos en forma de JSON.

Cuando teníamos la línea de código

```
app.post('/products', (req, res) => {
  res.send('Funciona el POST');
});
```

Hasta ahora solo usamos el parámetro **res** para devolver algo, resulta que el parámetro req nos sirve justamente para recibir información que viene en la llamada HTTP.

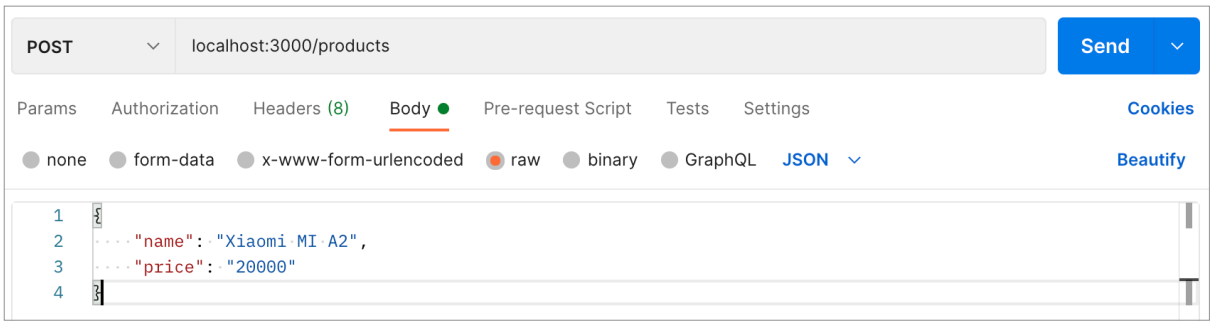
Antes de conocer como usar el parámetro req, lo que vamos a estar haciendo es que al recibir la información del lado del cliente como objeto JSON, lo vamos a parsear a un objeto para poder utilizarlo. Paras esto vamos a escribir

```
app.use(express.json());
```

Y ahora cada vez que recibamos información de tipo JSON, lo vamos a tener disponible como un objeto.

1. Vamos a hacer un request desde POSTMAN, enviando información de tipo JSON
2. Vamos a recibir esa información en el body del request.

1. Request desde POSTMAN con el body



2. Recibimos esa información desde el request.

```
app.post('/products', (req, res) => {
  console.log(req.body);
  res.send('Funciona el POST');
});
```

Como vemos, de esta manera se produce una conexión entre cliente y servidor, en donde el cliente envía información y el servidor la recibe.

Creando productos

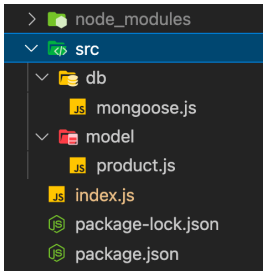
Ya teníamos en el index.js la posibilidad de crear productos, pero como mencionamos en clases anteriores, hay que mantener estructurado nuestro proyecto y por ahora el index.js (o app.js) va a ser el encargado de recibir consultas. La lógica la intentaremos mantener separada.

Nuestra estructura de carpetas es:

Proyecto

- Src
  - Db
    - mongoose.js
  - index.js

Ahora vamos a agregar una nueva carpeta dentro de src que se llame model y es donde vamos a empezar a guardar todos nuestros modelos. Para comenzar vamos a crear un archivo product.js y vamos a llevar toda la configuración de nuestro modelo hacia ese archivo. La carpeta quedará:



Dentro de db/mongoose.js solamente vamos a guardar la conexión a mongo:

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

mongoose.connect('mongodb://127.0.0.1:27017/ecommerce', {
  useNewUrlParser: true,
  useCreateIndex: true
});
```

Y dentro de nuestro product.js vamos a llevar todo lo relacionado con la creación del modelo de un producto.

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

const Product = mongoose.model('Product', {
  name: {
    type: String,
    validate(value) {
      if(value.length > 20) {
        throw new Error('El nombre tiene que tener menos de 20 caracteres')
      }
    }
  },
  price: {
    type: Number,
    required: true,
  }
});

module.exports = Product;
```

Teniendo nuestro custom module, simplemente queda importarlo en nuestro index.js, recibir la consulta del cliente, tomar los datos y guardar el nuevo producto.

```
app.post('/products', (req, res) => {
  const product = new Product(req.body);
  product.save().then(() => {
    res.send(product)
  }).catch(err => {
    res.send(err)
  });
})
```

Cuando todo va bien, simplemente vamos a responder el producto que se creó. Esto puede servir en el lado del cliente.

Cuando hay un error, lo que vamos a hacer es enviar el error que recibimos al intentar crear el producto… sin embargo hay algo que está faltando.

Errores y Status Code

Cada vez que hacemos una consulta http, tenemos distintos estados y códigos. Uno de los códigos de error más conocidos es el famosos 404 que significa “No encontrado” y es un error cuando el Back End no encuentra algo particular.

Para conocer todos los errores podemos ir a https://httpstatuses.com/ en donde tenemos un listado de todos los errores. Uno de los errores es el de 400 cuando alguien envía un request mal hecho (por ejemplo necesito que el cliente me mande el nombre del producto y no lo manda). Por ahora vamos a hacer pruebas enviando datos de forma errónea y devolveremos el error 400.

```
app.post('/products', (req, res) => {
  console.log(req.body);
  const product = new Product(req.body);
  product.save().then(() => {
    res.send(product)
  }).catch(err => {
    res.status(400).send(err)
  });
})
```

Verbo GET

Este verbo de uno de lectura y nos va a servir para leer todos los productos o productos específicos. Vamos a comenzar *fetcheando* la información de todos los productos que creamos.

Como venimos haciendo, pero sin darnos cuenta, el verbo que vamos a usar es .get, por lo tanto el método a ejecutar es el app.get();

```
app.get('/products', (req, res) => {
});
```

Mongoose nos provee un método (similar a mongo) que es el *.find()* y nos permite buscar elementos en la DB. Funciona por medio de promesas, por lo cuál vamos a buscar todos los productos, y luego de que esté ejecutamos un callback que me devolverá una lista de productos.

```
app.get('/products', (req, res) => {
  Product.find().then((products) => {
    res.send(products);
  })
});
```

Si tenemos un error, vamos a mandarlo con el catch.

```
app.get('/products', (req, res) => {
  Product.find().then((products) => {
    res.send(products);
  }).catch((err) => {
    res.status(500).send(err)
  })
});
```

Probamos este nuevo endpoint con Postman creando un GET request, *llamando* a la ruta de products con el verbo GET.

Buscar un producto particular por ID

Para buscar un producto por id, una de las formas que se utiliza es mandar un parámetro en el request por URL. Básicamente es poder llamar a un endpoint como

https://miweb.com/productos/1

Siendo el número 1 el id del producto que queremos buscar.

Para recibir ese parámetro, lo que hacemos es uso del objeto ***params*** que está dentro del request, es decir:

```
req.params
```

Sin embargo, cuando nosotros usamos una ruta como https://miweb.com/productos/1 no estamos definiendo un nombre de variable, sino que solo mandamos el número 1. La definición de ese nombre, la definimos en la ruta de nuestro servicio de la siguiente manera:

```
‘/product/:id'
```

Lo que estamos haciendo en este caso, es decir que cuando alguien entre a nuestra web /*product,* lo que viene después de la / se va a guardar en una propiedad id del objeto *params*. Entonces, todo junto:

```
app.get('/product/:id', (req, res) => {
  const _id = req.params.id;
});
```

Finalmente, mongoose nos provee la posibilidad de encontrar por ID con el método findById(); y funciona, como los métodos ya vistos, mediante promises.

```
  Product.findById(_id).then((product) => {
    res.send(product);
  }).catch(err => {
    res.status(404).send(err);
  })
```

Lo que tenemos que tener en cuenta es que es posible que no se encuentre un producto, y eso no va a devolver error. Entonces lo que podemos hacer es una pequeña validación para saber si hay o no hay usuario. Si no hay, devuelvo un error y sino devuelvo algo de éxito.

```
app.get('/product/:id', (req, res) => {
  const _id = req.params.id;
  Product.findById(_id).then((product) => {
    if(!product) {
      return res.status(404).send();
    }
    res.send(product);
  }).catch(err => {
    res.status(404).send(err);
  })
})
```

Actualizando un elemento
Verbo PATCH

Muchas veces vamos a necesitar actualizar propiedades de un elemento particular, y para eso utilizamos el verbo PATCH

Similar a lo que venimos haciendo vamos a crear el app.patch() con una ruta y con un callback que reciba req y res. Como vamos a cambiar un solo producto, vamos a recibir un id específico.

```
app.patch('/product/:id', (req, res) => {})
```

Vamos a recibir el id mediante params

```
const _id = req.params.id;
```

Y finalmente vamos a hacer uso de un método de mongoose que se llama “findByIdAndUpdate”. Este método recibe tres parámetros

```
Product.findByIdAndUpdate(_id, req.body, {new: true, runValidators: true})
```

1. Id
2. El objeto con las propiedades a actualizar (Si mandamos propiedades que no existen en el modelo, simplemente las ignora)
3. Un objeto de validación que en nuestro caso le decimos:
    1. Que devuelva el usuario original así lo tenemos en la consulta
    2. Que corra los validators al hacer el update.

En este caso vamos a utilizar la combinación de dos cosas:
1. El body: Para mandar el objeto a modificar
2. Los req.params: Para enviar el id del objeto a modificar.

Similar a los otros endpoints y métodos de mongoose:
1. Escuchamos cuando alguien le pega a nuestra ruta
2. Guardamos el Id
3. Ejecutamos el método disponible en el modelo, en este caso findByIdAndUpdate
4. Cuando esté preguntamos si hay algún producto actualizado
    1. Si no lo hay, envío error
    2. Si lo hay, envío el producto
5. Si hay un error de sistema, lo envuelvo en el catch y respondo error.

```
app.patch('/product/:id', (req, res) => {
  const _id = req.params.id;
  Product.findByIdAndUpdate(_id, req.body, {new: true, runValidators: true}).then((product) => {
    if(!product) {
      return res.status(404).send();
    }
    res.send(product);
  }).catch(err => {
    res.status(404).send(err);
  });
})
```

Eliminando un elemento
Verbo DELETE

¡Es el turno de ustedes!