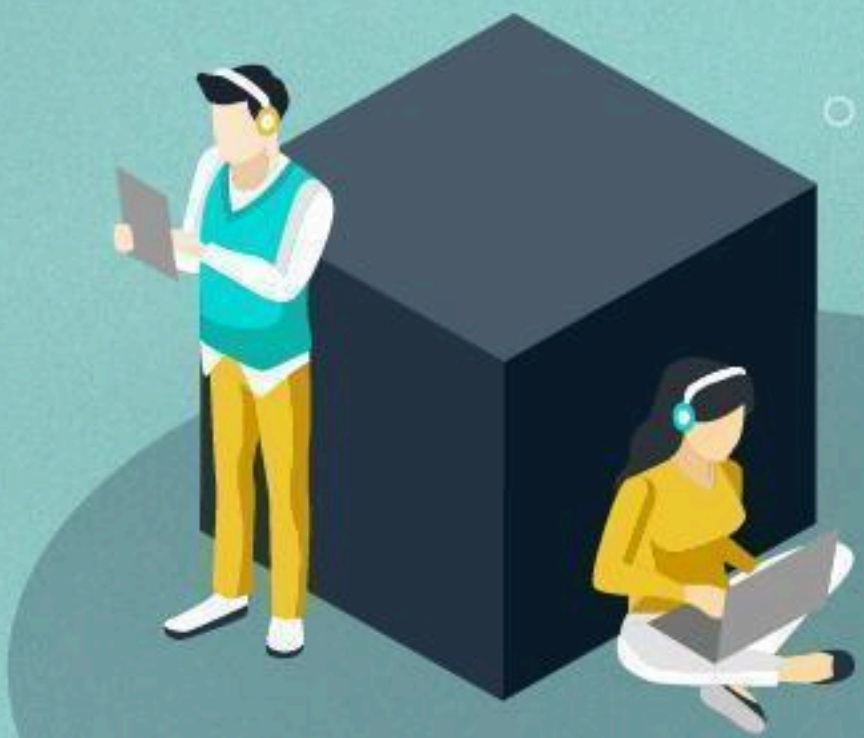
«Talento Tech»

Desarrollo Web 4

Clase 10



Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 10 | CRUD Express + Mongo

Temario:

- Preparación del entorno de trabajo
- Creación de un servidor en Express e Integración con mongoose
- Creación de nuestro CRUD en mongoose

Preparación del entorno de trabajo

En la clase de hoy vamos a estar haciendo un repaso de los contenidos vistos hasta el momento. Principalmente vamos a hacer un **CRUD** en Express js aplicando un modelo de base de datos no relacional (**Nosql**) utilizando un **ODM** que ya hemos visto que es mongoose.

⚠️💻 Recuerden que para poder iniciar este proyecto van a necesitar tener cuenta en mongo atlas. Si no lo hicieron aún les recomiendo que accedan al siguiente sitio:
https://account.mongodb.com/account/login

Una vez con la cuenta activada recuerden que tienen que tener las credenciales activas, estos temas han sido vistos en la Clase N°6.

Iniciando el proyecto

Para poder dar inicio a nuestro proyecto debemos crear una carpeta llamada *api-rest* o el nombre que deseen. Allí van a realizar los siguientes comandos para instalar las siguientes librerías:

Para crear nuestro archivo package.json:

```
npm init -y
```

Para instalar las dependencias:

```
nmp install express mongoose body-parser
```

Creación de un servidor en Express e Integración con mongoose

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Iniciamos el servidor

Una vez que tenemos ya la cuenta activa en **mongo atlas** y tenemos nuestro proyecto inicializado lo que tenemos que hacer ahora es crear un archivo **index.js** el cual nos permitirá poder alojar toda la información que va a contener nuestro servidor.

Lo primero que vamos a hacer en nuestro archivo es importar las dependencias de la siguiente manera :

Express como ya hemos visto nos servirá para poder crear nuestro servidor
Mongoose nos permitirá conectar nuestro servidor con la base de datos
Body-parser nos permitirá trabajar con formato Json

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
```

Creamos el Servidor

Para poder crear el servidor debemos implementar estas dos líneas de código :
En la app alojaremos en una variable nuestro servidor
Port será una variable que almacenará el puerto donde va a estar corriendo nuestro servidor

```
const app = xpress();
const PORT = 3000,
```

Establecer conexión entre Mongoose y express.js

A continuación podemos observar la forma de conexión que tiene **mongoose** implementando su método *".connect"*. En primer lugar deberemos ingresar la url que contiene las credenciales de nuestra base de datos de mongo atlas (podemos obtenerla de su sitio una vez ya logueados). Luego, nos aparecerá un objeto el cual brindará información

sobre el parseo de url y sobre cómo utilizar la topología unificada.

Debajo trabajamos el método como una **promesa**, ya que la conexión tiene su tiempo de latencia y puede ser exitosa como también puede fallar.

```
mongoose.connect('tu clave de mongo atlas', { useNewUrlParser: true,
useUnifiedTopology: true})
  .the(() => {
  })
  .catch((error) => {
     console.error('Error de conexión a la base de datos', error);
  });
```

Creando el Modelo

A continuación vamos a crear el **modelo** que será alojado dentro de nuestra **colección** que vamos a especificar a continuación:

```
const serieSchema = new mongoose.Schema ({
  title: String,
  genre: String,
  releaseYear: Number,
});
```

Hasta ahora solo creamos el **modelo**, el cual debemos establecer una **colección** en donde se almacenará:

```
const Serie = mongoose.model('Serie', serieSchema);
```

En esta línea de código podemos ver que el tipo de modelo serieSchema se almacenará en **mongo atlas** en la colección ‘Serie’.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Incorporación de Middlewares

A continuación trabajaremos con la incorporación de un **middleware** que nos va a poder permitir procesar nuestras solicitudes **'http'**:

```
app.use(bodyParser,json());
```

Creación de nuestro CRUD en Mongoose

POST

Una vez que ya tenemos seteada toda la configuración principal vamos a empezar a trabajar con los métodos para poder trabajar el concepto de Lectura, Alta , Baja y Modificación.

Primero utilizaremos el método post el cual nos servirá para hacer la carga de elementos a nuestra base de datos :

```
app.post('/series', async (req, res) => {
    try {
        const newSerie = await Serie.create(req.body);
    } catch (error) {
        res.status(500).json({ error: 'Error al crear la serie.' });
    }
});
```

Como ya hemos visto antes todos los métodos que realizan solicitudes **'http'** en **Express** requieren de dos argumentos. El primero es la ruta a la cual irá y la segunda viene a ser la función controladora que van a ejecutar las acciones que se realicen cuando asistimos a ese endpoint.

Como verán en este caso, estamos usando **try & catch**, el primero lo usamos para que intente realizar la solicitud obteniendo la información del objeto request.body siendo enviado por parámetro mediante el método **'create'**. que nos permitirá crear ese elemento. Y en el segundo caso, devuelve un error de tipo 500 en el caso de no poder crearse.

GET

Luego crearemos el endpoint para traer las series que se hará mediante el método **get.** Continuamos implementando **try & catch** para la solicitud y captura de errores. En el método try realizaremos la solicitud implementando el método **find().**

```
app.get('/series‘, async (req, res) => {
   try {
      const series = await Serie.find();
      res.json(series);
   } catch (error) {
      res.status(404).jason({ error: 'Error al obtener las series.'});
   }
});
```

Actualización de una Serie

```
app.put('/series/:id', async (req, res) => {
   try {
      const updateSerie = await Serie.findByIdAndUpdate(req.params.id,
   req.body, { new: true });
      res.jason(updatedSerie);
      } catch (error) {
         res.status(500).json({ error: 'Error al actualizar la serie.' });
      }
});
```

En este caso estaremos trabajando con la actualización de series, por ende podemos observar que tenemos un /series/:id . El “ :id” lo usamos para poder tener una url dinámica y poder capturar su param y poder enviárselo a mongoose para que realice su magia.

En este caso el id no va a ser autoincremental como en **SQL** sino es un **UUID** (único id) que se trabaja de forma aleatoria.

Luego lo que hacemos en este caso es en una constante realizar la operación la cual va a buscar el elemento acorde a su id y luego lo va a modificar. El primer parámetro es el de la url, el segundo es la request que solicitamos (recuerden redactar un objeto Json que respete el modelo ya creado), el último parámetro indica que debe retornar el documento actualizado.
Por último haremos una realizamos la respuesta devolviendo lo que nos retorna la variable "updatedSerie". En el caso del "catch" devolvería un 500 como error.

DELETE
Por último este método como ya hemos visto antes nos permitirá eliminar elementos de la base de datos.
A continuación veremos un código de muestra:
Dicho código, lo que hará, será obtener un id por param y luego se implementará el método "*findByIdAndDelete*" que recibe como parámetro solo el id de la url que nosotros hagamos. En el caso de que la promesa resulte exitosa devolverá una respuesta Json con el objeto eliminado. En el caso de fallar devolverá un error en este caso de status 500.

```
app.delete('/series/:id', async (req, res) => {
    try {
        const deletedSerie = await Serie.findByAndDelete(req.params.id);
        res.json(deletedSerie);
    } catch (error) {
        res.status(500).json({ error: 'Error al eliminar la serie.'});
    }
});
```

Configuración del puerto del servidor

Por último realizaremos la configuración del puerto para poder terminar nuestro **servidor**:

```
app.listen(PORT, () => {
    console.log(`Servidor iniciado en http://localhost:${PORT}`)
});
```

De ésta forma nuestro proyecto queda terminado podemos realizar pruebas en **postman** para poder ver si funciona.

Pueden usar cualquiera de estos ejemplos para hacer el post recuerden hacerlo a http://localhost:3000/series especificando que van a implementar un **POST**.

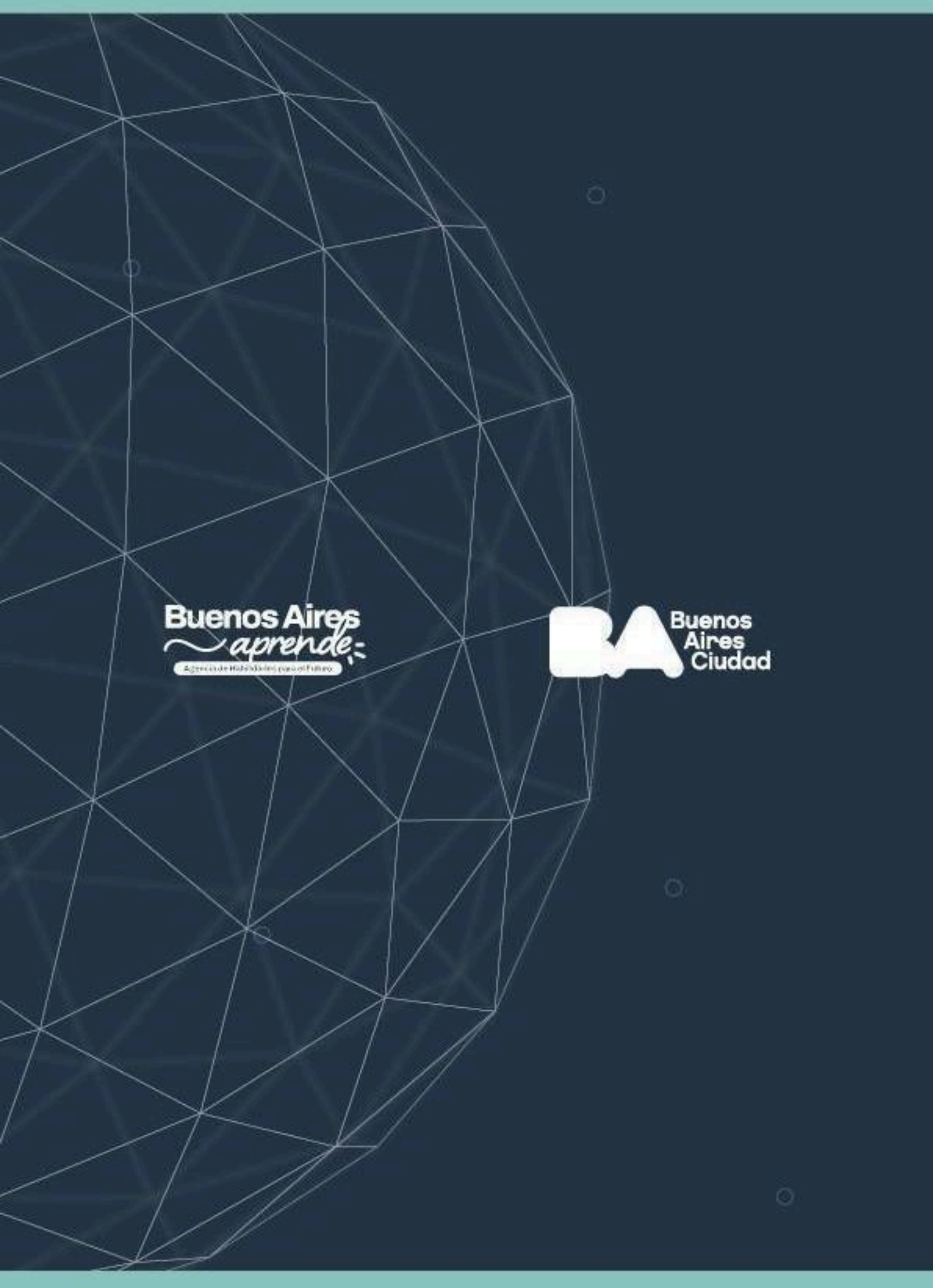
A continuación tienen 4 modelos para usar, para poder probarlo:

```
{
        "title": "Breaking Bad",
        "genre": "Drama",
        "releaseYear": 2008
    },
    {
        "title": "Stranger Things",
        "genre": "Sci-Fi",
        "releaseYear": 2016
    },
    {
        "title": "The Mandalorian",
        "genre": "Action",
        "releaseYear": 2019
    },
    {
```

```
    "title": "Game of Thrones",
    "genre": "Fantasy",
    "releaseYear": 2011
}
```

Luego, les sugiero que implementen el método **GET** para conocer los **ID** de cada elemento de la **colección** y, por último, pueden implementar los otros verbos pegando el id donde iría ":id" .
Ejemplo: http://localhost:3000/:id donde dice ":id" ahí deberían sustituirlo por el id que les de la base de datos del objeto que quieran **modificar** o **eliminar**. Recuerden que para modificar una serie, debemos mandar el body de la request como Json en Postman.

Y con eso tendríamos ya nuestro proyecto funcionando.


Buenos Aires
aprende
Buenos
Aires
Ciudad