Curso de Back-End con Node.js (Avanzado)

Clase 04

Temario

Temario

- SQL vs. NoSQL.
- Más MongoDB
  - `find`.
  - `findOne`.
  - `findById`.
- Ejercicios.

SQL vs. NoSQL

SQL vs. NoSQL – Ver video.

	SQL	NoSQL
Tipo	Relacional - Basada en tablas que contienen registros (filas).	No relacional - Basada en colecciones que contienen documentos. También puede ser clave-valor, grafos, etc.
Estructura	Definida - Esquema rígido.	Dinámica (sin esquemas).
Redundancia	Se diseñan para tener la menor redundancia posible.	Permite tener redundancia de datos.
Casos de uso	Para aplicaciones que requieren datos consistentes (ej: instituciones financieras).	Para aplicaciones que requieren grandes cantidades de datos.
Escalabilidad	Vertical (aumentar prestaciones del servidor).	Horizontal (agregar más servidores).

.find

Buscar documentos: `.find()` (1/8)

Para encontrar documentos en MongoDB, dado un modelo, se utiliza el método `find`.

Como primer parámetro, `find` recibe un objeto de condiciones.

Si el objeto es vacío, la query responderá con todos los documentos (dado que no hay condiciones).

```
Article.find({}); // Retorna todos los documentos
```

Buscar documentos: `.find()` (2/8)

Si necesitamos realizar una búsqueda que se acople a ciertos criterios, podemos configurar el objeto para refinar los resultados.

```
Article.find({ title: "JavaScript" });
```

Retorna los documentos cuyo título sea exactamente igual a "JavaScript".

Buscar documentos: `.find()` (3/8)

Para aquellos que sepan expresiones regulares, ¡Mongoose también las soporta!

```
Article.find({ title: /hack/i });
```

Retorna todos los documentos cuyo título *matchee* con una expresión regular, en este caso, 'hack' case-insensitive.

Buscar documentos: `.find()` (4/8)

Como segundo parámetro, se puede agregar un *string* con aquellos atributos que se quieren incluir en la query.

Para esto se suele usar la palabra **selección**.

En caso de omisión, se incluirán todos los campos.

```
Article.find({ title: "Node.js" }, "content author");
```

Retorna todos los documentos cuyo título sea exactamente “Node.js”, y de esos documentos solo “traerán” los atributos `content` y `author`.

Buscar documentos: `.find()` (5/8)

También se pueden seleccionar campos por exclusión:

```
Article.find({ title: "Node.js" }, "-author");
```

Retorna todos los documentos cuyo título sea exactamente “Node.js” y de esos documentos se incluirán todos los atributos menos `author`.

Buscar documentos: `.find()` (6/8)

Como tercer parámetro, se puede pasar un objeto de opciones para la query.

En este objeto podemos incluir opciones como:

```
Article.find({ title: "Academy" }, null, { skip: 10 });
```

Retorna todos los documentos cuyo título sea exactamente sea “Academy” y se saltean los primeros 10 resultados. ¡Útil para construir una paginación!

Buscar documentos: `.find()` (7/8)

Otra operación muy útil es la de ordenar.

La regla es: en el campo `sort`, se pasa un objeto con *key* siendo el nombre del campo y *value* el tipo de orden (1 para ascendente, -1 para descendente)

```
Article.find({ title: "Hack" }, null, { sort: { likes: 1 } });
```

Retorna todos los documentos cuyo título sea exactamente “Hack”, ordenados ascendentemente por cantidad de `likes` (siendo éste un atributo del modelo).

Buscar documentos: `.find()` (8/8)

Otra forma de configurar la query es concatenando los métodos `select()` y `setOptions()` para cada uno de estos campos. Ambas son equivalentes y a gusto de cada programador.

```
Article.find({ title: "Hack Academy" })
    .select('-author')
    .setOptions({ sort: { title: 1 } });

// O también se puede escribir de esta forma:
Article.find({ title: "Hack Academy" }, '-author', { sort: { title: 1 } });
```

.findOne

Buscar un documento específico: `.findOne()`

Hay veces que necesitamos un documento específico, en lugar de varios. Para ello usamos `findOne`.

También soporta selectores, pero cuando se trata de opciones, sólo soporta una: `lean`. Cuando esta opción es `true` en lugar de retornar un *documento de Mongoose*, lo convierte a objeto de JavaScript.

```
Article.find({ title: "Hack Academy" })
    .select('-author')
    .lean(true);

// O también se puede escribir de esta forma:
Article.findOne({ title: "Hack Academy" }, '-author', { lean: true });
```

.findById

Buscar un documento específico: `.findById()`

Habíamos visto que al guardar un documento en una colección, MongoDB automáticamente le asigna un identificador único `_id`.

Si necesitamos buscar por ese valor, la buena práctica es usar `findById(algunId)` en lugar de hacer `findOne({ _id: algunId })`.

¿Por qué? ¿Cuál es la diferencia?
La diferencia está en cómo se comportan con al pasarles el valor: `undefined`.

`findOne(undefined)` y `findOne({ _id: undefined })` son equivalentes a hacer `findOne({})`. Las tres queries van a traer un elemento cualquiera. Sin embargo, Mongoose traduce `findById(undefined)` a `findOne({ _id: null })`, y por lo tanto no traerá un elemento cualquiera. En lugar de eso, no traerá nada.

ObjectId

ObjectId – Trivia previa

Sin probarlo en la consola, ¿cuál es el resultado de esta comparación?

```
{a: 1} === {a: 1}
```

Estos dos objetos, ¿son exactamente iguales?

ObjectId

La respuesta anterior es `false`. ¿Por qué? Porque los objetos son referencias.

Y `ObjectId` es un objeto. Es decir que dado un documento proveniente de MongoDB, el valor de la clave `_id` será de un **objeto**. Por esto, para comparar id's de documentos es necesario convertirlos a string y recién ahí compararlos.

```
const idOfArgentina = "5eebf017feed2b631431b221";
const argentina = Team.findById(idOfArgentina);

argentina._id === idOfArgentina; // false
argentina._id.toString() === idOfArgentina; // true
```

Ejercicio

Ejercicio (1/3)

👉 Para ahorrar tiempo, pueden **duplicar** la carpeta con el proyecto de la última clase.

1. Crear un nuevo directorio con el nombre `ha_node2_clase04`.
2. Crear un nuevo proyecto de Node con el comando: `npm init -y`.
3. Instalar como dependencias `express` y `mongoose`.
4. Organizar el proyecto Express como aprendimos en clases anteriores, con los archivos `server.js`, `routes.js` y las carpetas `models` y `controllers`.
5. Crear el modelo `Team` con los atributos `code`, `name` y `flag` (todos strings). Los dos primeros deben ser obligatorios.

Ejercicio (2/3)

6. Crear tres *endpoints*:

- `GET /teams` que responda con todos los equipos.

  EXTRA: agregar la posibilidad de que a través del *query string* se puedan establecer 3 parámetros: `sortBy`, `order` y `skip`; de forma tal que la *response* se ajuste a dichos parámetros.

- `GET /teams/:code` que responda con el equipo correspondiente.

  EXTRA: agregar la posibilidad de que a través del *query string* se puedan establecer 2 parámetros: `include`, `exclude`, que incluyan o excluyan atributos del equipo.

- `POST /teams` que cree el equipo que incluya el *body* del *request*.

Ejercicio (3/3)

7. Crear un archivo `teamSeeder.js` que tenga el contenido de este gist. Colocarlo en una carpeta llamada `seeders` (en la raíz del proyecto).

8. Modificar `teamSeeder.js` de forma tal que popule la colección `teams` (en MongoDB) con cada uno de los objetos que se encuentran en el array.

9. Ejecutar el archivo `teamSeeder.js`. Esto se deberá realizar una única vez para evitar contenido duplicado en la base de datos. utilizar la función la función `mongoose.connection.dropDatabase()` para borrar el contenido de la base de datos antes de ejecutar el *seeder*.