

# Clase 12: Mongoose

Sitio: [Centro de E-Learning - UTN.BA](#)  
Curso: Curso de Backend Developer - Turno  
Noche  
Libro: Clase 12: Mongoose

Imprimido  
por: Nelson Brian Avila Solano  
Día: Tuesday, 9 de December de 2025,  
19:56

# Tabla de contenidos

1. Introducción
2. Schema Types
3. Subdocumentos
4. Validators

# 1. Introducción

En esta unidad exploraremos los conceptos fundamentales de Mongoose, una biblioteca para MongoDB que permite modelar datos de manera flexible en aplicaciones basadas en Node.js. Veremos cómo se estructuran los schemas, qué tipos de datos pueden utilizarse y cómo se aplican validaciones, índices y modificadores para optimizar la gestión de la información.

Además, abordaremos técnicas avanzadas como el uso de subdocumentos, la relación entre colecciones mediante populate, y la implementación de JSON Web Tokens (JWT) para la autenticación y seguridad en aplicaciones web. Finalmente, analizaremos cómo Mongoose facilita la validación de datos y la encriptación de contraseñas con herramientas como Bcrypt, garantizando una gestión segura y eficiente de la información en bases de datos NoSQL.

## 2. Schema Types

### Schema Types

#### Mongoose - Schema type

Los tipos de datos que se puede declarar en el schema son los siguientes:

- String
- Number
- Date
- uffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

#### Mongoose – Schema type (Índices)

- index:** boolean, establece un indice con el campo especificado.
- unique:** boolean, define un indice de tipo unique

```
var schema2 = new Schema({
  test: {
    type: String,
    index: true,
    unique: true // Unique index. If you specify `unique: true`
    // specifying `index: true` is optional if you do `unique: true`
  }
});
```

#### Mongoose – Schema type (String)

- lowercase:** Se aplica minúsculas a toda consulta
- uppercase:** Se aplica mayúsculas a toda consulta
- trim:** Quita espacios al elemento aplicado en las consultas
- match:** Valida la expresion regular especificada

- enum**: Crea un validador en base al array especificado
- minlength**: Valida que el numero de caracteres sea mayor al especificado
- maxlength**: Valida que el numero de caracteres sea menor al especificado

```
const schema1 = new Schema({ name: String }); // name will be cast to string
const schema2 = new Schema({ name: 'String' }); // Equivalent

const Person = mongoose.model('Person', schema2);
```

Se puede declarar el tipo con la clase “String” o como ‘String’

El metodo toString() permite convertir a string cualquier valor (excepto arrays)

## Mongoose – Schema type (number / date)

- Min**: Valida que el numero ingresado sea mayor al establecido
- Max**: Valida que el numero ingresado sea menor al establecido

```
const schema1 = new Schema({ age: Number }); // age will be cast to a Number
const schema2 = new Schema({ age: 'Number' }); // Equivalent

const Car = mongoose.model('Car', schema2);
```

Se puede declarar el tipo con la clase “Number” o como ‘Number’

## Modificadores

### Set

Por ejemplo tenemos el campo sitioweb que deberia comenzar con ‘http://’ o con ‘https://’, pero en lugar de forzar al cliente a agregar esto en la UI, puedes escribir un modificador personalizado que valide la existencia de estos prefijos y los agregue cuando sea necesario. Para agregar tu modificación personalizado necesitaras crear el nuevo campo sitio web con una propiedad set.

Cada usuario creado tendrá una url de un sitio web bien formada que se modifica en tiempo de creacion.

## Get

Los modificadores getter se usan para modificar los datos existentes antes de enviar los documentos a la siguiente capa. Por ejemplo en nuestro ejemplo previo un modificador getter a veces seria mejor cambiar el documento de usuario ya existiente modificando su campo sitioweb en tiempo de busqueda, en lugar de recorrer toda la coleccion MongoDB actualizando cada componente.

Con el método `UserSchema.set('toJSON', {getters: true});` habilitamos para que todas en todos los json devueltos por las consultas se aplica el getter

## Atributos Virtuales

Algunas veces puedes querer tener propiedades de los documentos calculadas dinamicamente, las cuales no estan realmente presentes en el documento. A estas propiedades se le llaman atributos virtuales y se pueden usar para obtener requisitos comunes.

Por ejemplo digamos que quieres agregar un nuevo campo `nombreCompleto`, que represente la concatenacion del nombre y del apellido del usuario. Para ello usaremos el metodo `virtual()`

Retornara en el json un nuevo atributo del documento denominado “nombreCompleto”

Pero los atributos virtuales pueden tambien tener setters para ayudar a tus documentos como prefieras en lugar de solamente agregar mas atributos. En este caso digamos que quieres romper la entrada del campo `nombreCompleto` en sus campos `nombre` y `apellido`.

## Índices

### Mongoose – Modificadores propios

MongoDB soporta varios tipos de índices para optimizar la ejecución de las búsquedas. Mongoose también soporta la funcionalidad de indexado e incluso nos permite definir índices secundarios.

El ejemplo básico de indexación es el índice único, el cual valida la unicidad de un campo de un documento en una colección. En nuestro ejemplo es común que los nombres de usuario sean únicos, así que vamos a modificar la definición de `UsuarioSchema` para realizar esto



Mongoose también soporta la creación de índices secundarios usando la propiedad `index`. Así si por ejemplo sabes que tu aplicación tendrá muchas búsquedas que conlleven al campo `email`, podrás optimizar estas búsquedas creando un índice secundario `email`



## Populate

Imaginemos una base de datos relacional de libros. Tendríamos una tabla con los títulos de los libros y otra con los datos de los autores. El campo `autor` en la tabla de libros, apuntaría a un ID o clave primaria de un autor de la tabla `autores`.



supongamos un modelo sencillo para libro de la siguiente manera:

Si nos fijamos, para el campo autor en el modelo libro hemos usado el tipo `Schema.ObjectId` y la referencia al modelo Autor. Esto nos permitirá establecer la relación entre un campo de una tabla y otra.

Para consultar los datos lo haremos de la siguiente manera:



Luego aplicamos método `populate` dentro del callback de libros

Luego aplicamos método `populate` dentro del callback de libros



La línea `Autor.populate(libros, {path: "autor"},...)`; toma el array de objetos libros y le indica que en la ruta autor lo "popule" con los datos del modelo Autor. Quedando una respuesta más completa como este ejemplo:

En el caso de querer guardar datos se debe asociar al campo autor de un libro un object id valido, por ejemplo:

Utilizando `async / await`

## 3. Subdocumentos

### Subdocumentos

Permite definir un esquema dentro de otro, es decir un documento hijo dentro de un documento padre.

### Definición Schema

### Insertar datos

Se hace un push sobre el subdocumento relacionado.

**Leer un subdocumento.** Aplicando un find para consultar todos los productos (documentos), el detalle del subdocumento se visualiza de la siguiente manera

### Queries



## findbyIdandUpdate

Ver todos los queries

<https://mongoosejs.com/docs/queries.html>

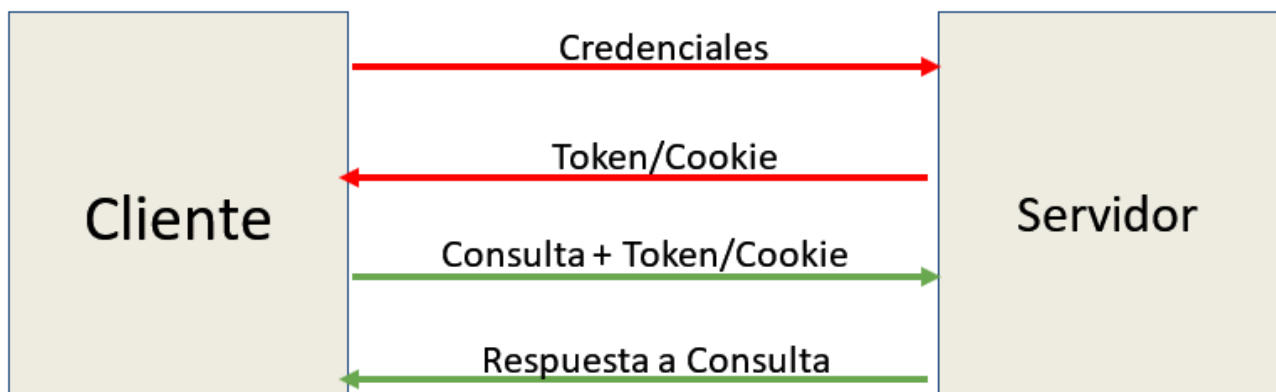
## Validations

## JWT

## Autenticación y Autorización

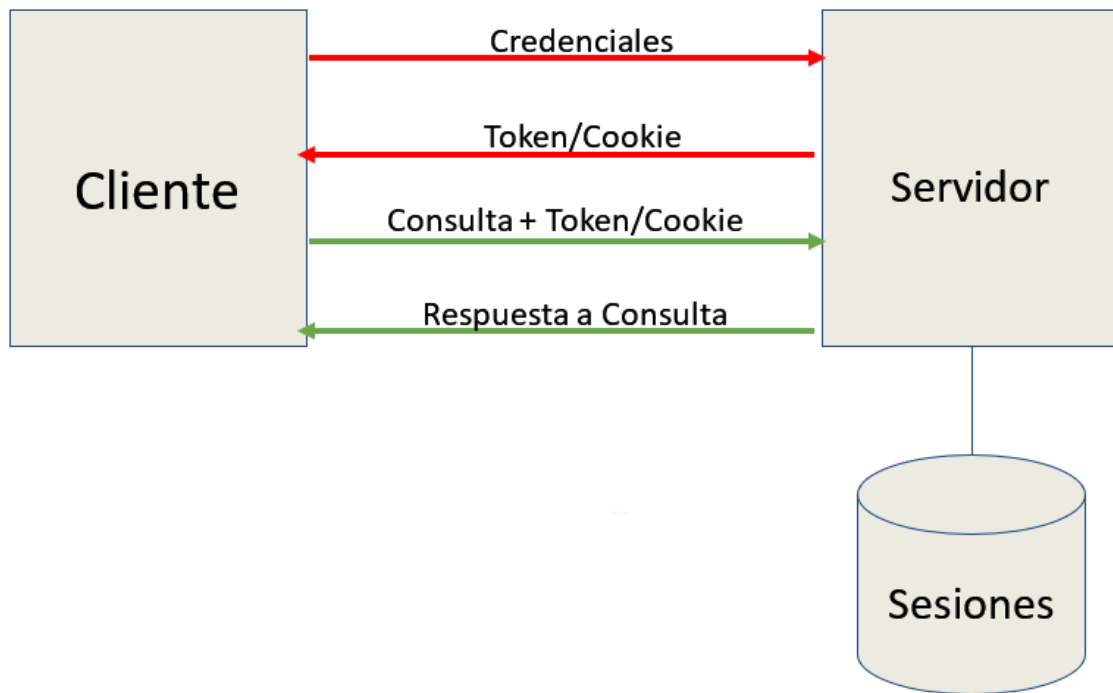
Definición de conceptos

- **Autenticación:** Proceso para identificar que “algo” o “alguien” es quien dice ser.
- **Autorización:** Proceso por el cual se determina la posibilidad de obtener algún tipo de información.



## Métodos

## Sesiones

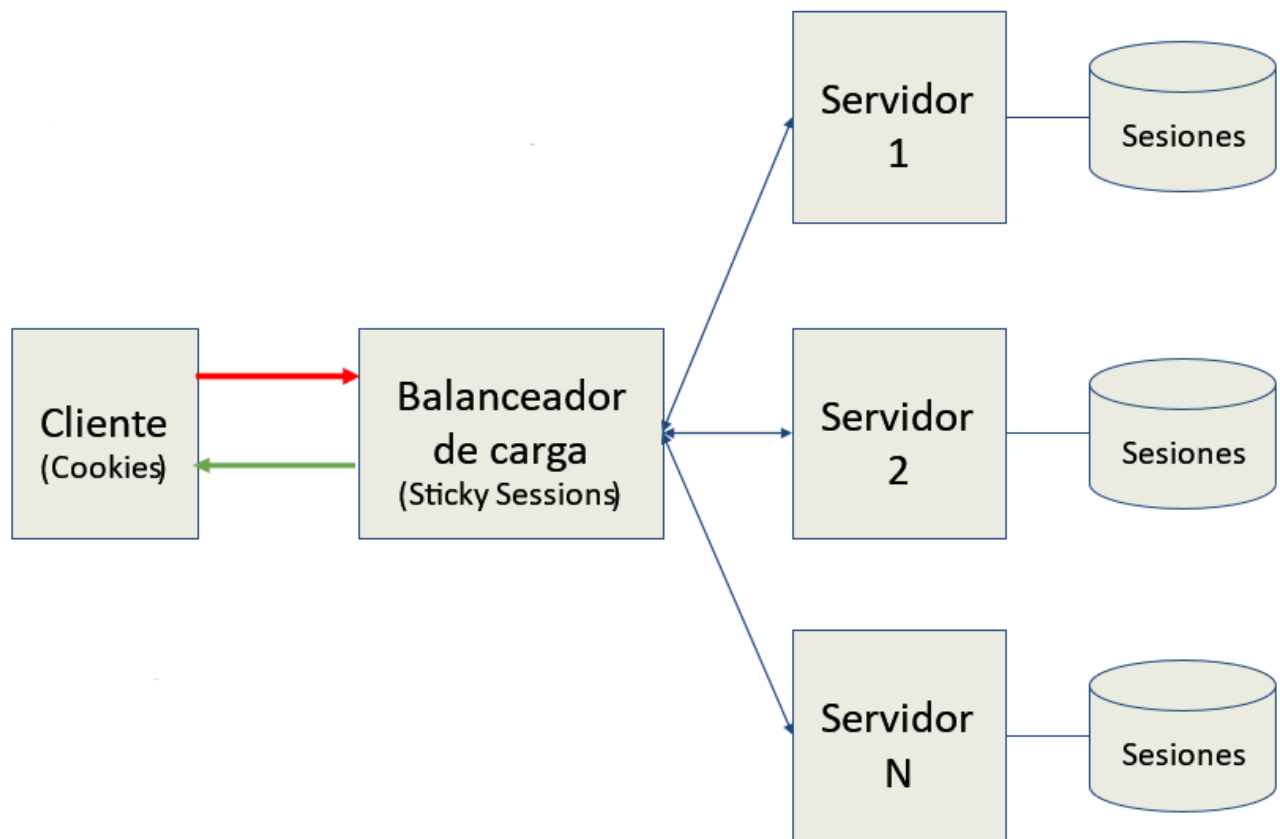
**Ventajas**

- Las tecnologías para servidores soportan sesiones de forma nativa
- Fácil de usar e implementar

**Desventajas**

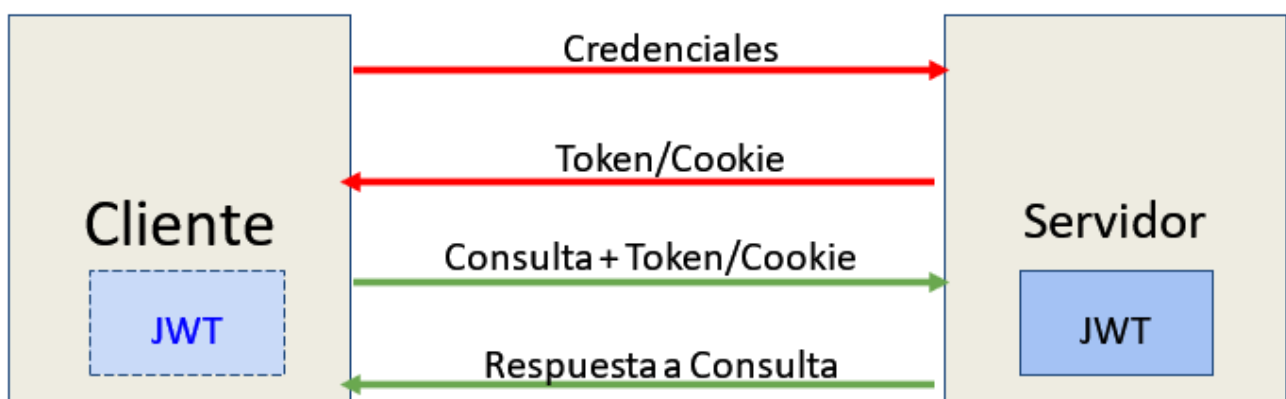
- No es eficiente para sistemas distribuidos

**Sesiones / problema**



En un ámbito de producción donde tengo varios servidores, este esquema es poco eficiente

## JWT



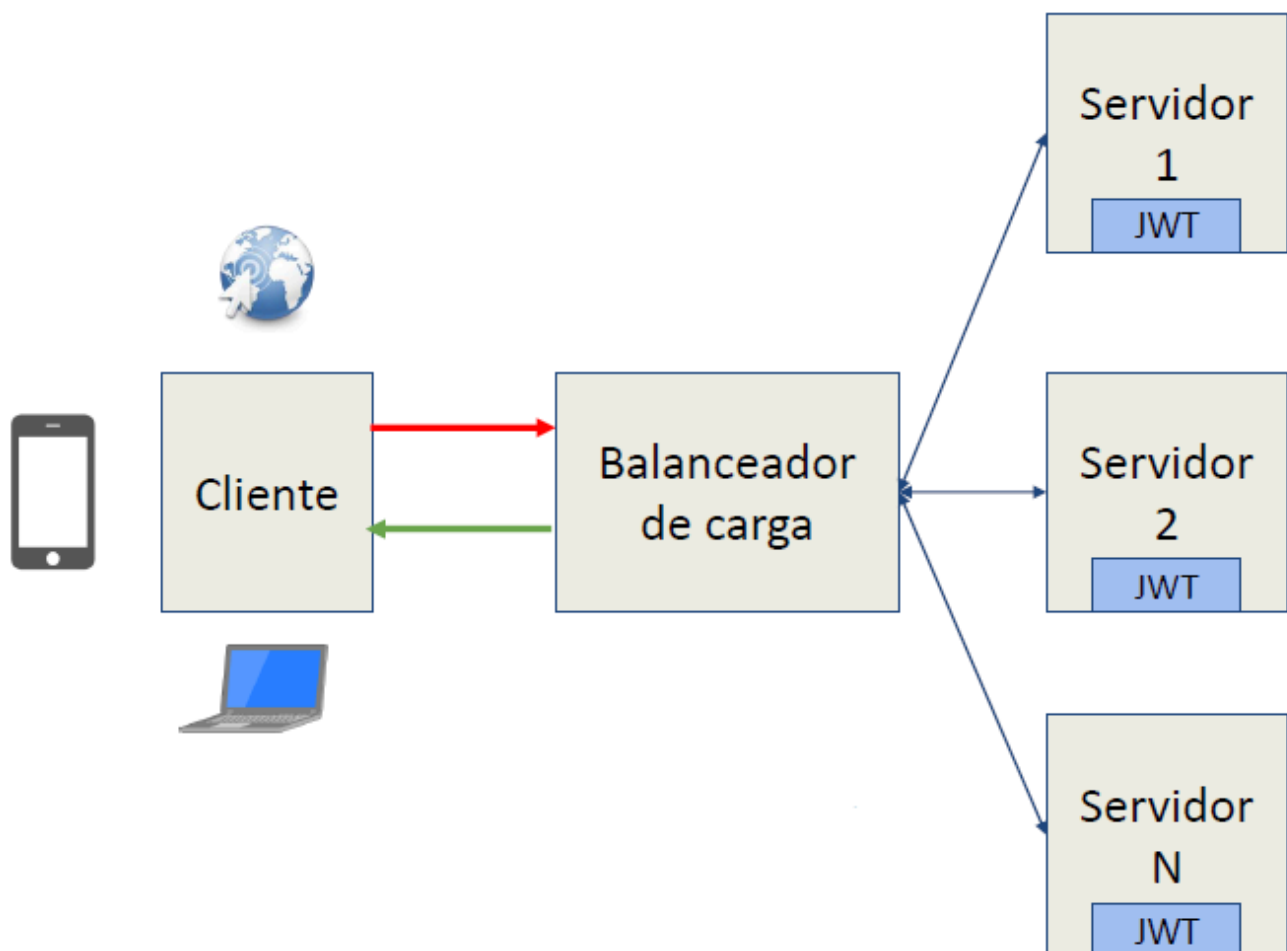
### Ventajas

- No existe el manejo de sesiones, cada request es independiente.
- Independencia entre plataformas

### Desventajas

- Cross Domain y CORS

## JWT / solución



La utilización de un token con “información autocontenida”, es más eficiente en arquitecturas distribuidas, además que permite independencia en las plataformas cliente.

## JSON WEB TOKEN

### ¿Que es JWT?

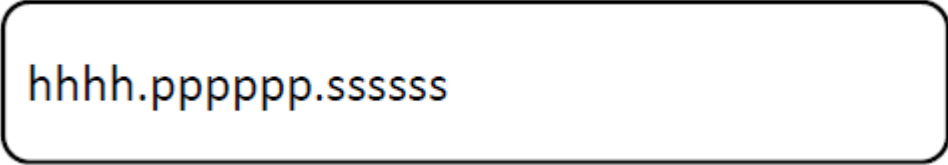
JSON Web Token (**JWT**) es un **standard abierto** (RFC 7519) que **define** una forma **compacta** y **autocontenida** para la **transmisión** segura de información entre partes por medio de un objeto **JSON**.

Esta información puede ser verificada y validada porque se encuentra firmada digitalmente usando una clave secreta.

*Adicionalmente, los JWT pueden ser encriptados para ocultar que la información contenida sea inaccesible.*

### JSON:

- Header
- Payload
- Signature



hhhh.pppppp.ssssss

## Header

El header típicamente contiene dos partes: El tipo de token, que es

JWT y algoritmo de hash que se utiliza para firmar el token, como HMAC, SHA256 o RSA.

Ejemplo:

## Payload

La segunda parte del token es el payload, que contiene los “claims”. Los “claims” son valores acerca de la entidad (generalmente el usuario que se está logueando) y además puede contener datos adicionales.

Existen 3 tipos de “claims”:

- **Registered**

**iss** (issuer), **exp** (expiration time), **sub** (subject), etc.

- **Public**

- **Private**

## JWT / Ejemplo

## Librerías

## Instalación NodeJs

## Uso básico

### NodeJs - JWT / Crear nuevo token

- El “claim” exp, define el tiempo de expiración del token
- El “claim” data es privado y puede utilizarse para pasar cualquier tipo de información, incluso otro JSON.

### NodeJs - JWT / Verificar un token

## BCRYPT

### JWT - Bcrypt

Modulo utilizar para encriptar el password . Debemos ejecutar por consola:

**npm install bcrypt**

## Aplicación

App.js

Se incluye modulo de jwt

Se define secret key. set(“secretKey”,”nodeAPI”)

A los métodos privados (que solo pueden ser accedidos por usuarios autenticados) se les aplica middleware para validar su token. Metodo validateUser



## **Route de autenticación**

### **Controller de autenticación**

#### **Require de modulos**

- **bcrypt**
- **jwt**
- **authenticationModel**

#### **Registro de usuarios. Aplica método create de mongoose**

**Login de usuarios. Si el usuario existe y el password es correcto (comparado con bcrypt porque esta encriptado) entonces se genera token**

#### **Model de autenticación**

**Se aplica middleware de tipo “pre” --> save ”. Toma el password sin encriptar y lo encripta para ser almacenado**

## 4. Validators

### Validators

Mediante la definición del esquema se pueden validar los campos a insertar:



En este caso validamos que el tipo de dato insertado sea un string



**En el campo usuario vemos aplicado:**

- **Unique:** no permite insertar otro documento con ese campo repetido
- **Required:** Es un campo obligatorio. El parámetro puede ser
  - required: true (valida que es obligatorio, en caso de error se muestra mensaje por defecto)
  - required: [true, msj] (permite personalizar el mensaje)

**En el campo usuario vemos aplicado:**

- **Minlength:** N (Valida que el campo tenga al menos n caracteres, en caso de error muestra mensaje por defecto)
- **Minlength:** [n,msj] (personaliza el mensaje en caso de error)

Para maxlength sigue la misma regla

También se puede aplicar una validación personalizada El método `validate` tiene 2 índices:

- **Validator:** Método que se aplicara para validar el campo (reglas de validación)
- **Message:** Mensaje renderizado en caso de error

Para ver mas acerca de métodos de validación: <https://mongoosejs.com/docs/4.x/docs/validation.html>

## Queries

Sobre los métodos de búsqueda, por ejemplo `find` se pueden aplicar queries para filtrar información, ordenar o bien seleccionar elementos del resultado a visualizar. Ejemplo

*Se aplica el orden por precio de forma ascendente sobre los resultados devueltos por el método `find`.*

*En caso de aplicar `Price: -1` el orden será descendente*

*Aplica el `select` de los atributos especificados (en el ejemplo*

*`Price y name`), de esta forma solo se retornaran dichos datos.*

*En este caso se aplica un `where` para filtrar por igualdad por campo `name`*

Para ver mas acerca de queries:

<https://mongoosejs.com/docs/4.x/docs/queries.html>

## Mongoose Paginate

Mongoose paginator es un modulo que nos permite paginar nuestras consultas.

Para su utilización debemos instalando ejecutando:

**`npm install mongoose-paginate-v2`**

En el archivo `bin/mongodb.js` aplicar lo siguiente:

*El campo `limit` se aplica por defecto a todas las consultas.*

Luego en el modelo en el cual queramos aplicar el paginado, configurar lo siguiente (ejemplo ProductsModel)

Por ultimo al realizar las consultas debemos aplicar el método paginate (ejemplo controller/products.js)

1er parámetro ({}): Permite filtrar documentos, al igual que find 2 do parámetro ({populate...}): son las opciones aplicadas al paginado

## Opciones

- Select: Selección de ciertos atributos
- Sort: Orden en los resultados
- Populate: Aplica el populate en base al modelo especificado
- Offset
- Page
- Limit

Se puede encontrar mas información en: <https://www.npmjs.com/package/mongoose-paginate>

## DOTENV

Modulo que nos permite manejar con mayor facilidad la configuración de nuestras variables de entorno

Instalar el modulo ejecutando:

```
npm install dotenv
```

**Crear un archivo llamado .env en la raíz de nuestro proyecto:**

En dicho archivo declarar las variables de entorno:

Hacer uso de dichas variables, por ejemplo para la definición del secretKey en el app.js

*A la variable definida se le antepone **process.env***

## Nodemailer

Nodemailer es un modulo utilizado para node que nos permite realizar el envío de emails.

Para instalar ejecutamos:

**Npm install nodemaile**

Luego creamos el archivo email.js dentro del directorio bin

Bin/email.js

*En caso de configurar una cuenta de otro cliente distinto a Gmail, se debe hacer de la siguiente forma*

Bin/email.js

controller/authentication.js

Para ver mas acerca de nodemailer:

<https://nodemailer.com/>