

Clase 12: Mongoose

Sitio: [Centro de E-Learning - UTN.BA](#)

Curso: Curso de Backend Developer - Turno
Noche

Libro: Clase 12: Mongoose

Imprimido

por:

Nilo Crespi

Día:

Friday, 23 de January de 2026,
10:23

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
- Buffer
- 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)

```
new Person({ name: 42 }).name; // "42" as a string
new Person({ name: { toString: () => 42 } }).name; // "42" as a string

// "undefined", will get a cast error if you `save()` this document
new Person({ name: { foo: 42 } }).name;
```

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 personalizada necesitaras crear el nuevo campo sitio web con una propiedad set.

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

```
var UsuarioSchema = new Schema({
  nombre: String,
  apellido: String,
  email: String,
  usuario: {
    type: String,
    trim: true
  },
  password: String,
  creado: {
    type: Date,
    default: Date.now
  },
  sitioweb: {
    type: String,
    set: function(url){
      if(!url){
        return url;
      } else {
        if(url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0){
          url = 'http://' + url;
        }
        return url;
      }
    }
  }
});

mongoose.model('Usuario', UsuarioSchema);
```

Get

Los modificadores getter se usan para modificalos 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 existenque modificando su campo sitioweb en tiempo de busqueda, en lugar de recorrer toda la colección 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

```
var mongoose = require('mongoose'),  
    Schema = mongoose.Schema;  
  
var UsuarioSchema = new Schema({  
    nombre: String,  
    apellido: String,  
    email: String,  
    usuario: {  
        type: String,  
        trim: true  
    },  
    password: String,  
    creado: {  
        type: Date,  
        default: Date.now  
    },  
    sitioweb:{  
        type: String,  
        get: function(url){  
            if(!url){  
                return url;  
            } else {  
                if(url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0){  
                    url = 'http://' + url;  
                }  
                return url;  
            }  
        }  
    }  
});  
  
UserSchema.set('toJSON', {getters: true});
```

Atributos Virtuales

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

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

```
UsuarioSchema.virtual('nombreCompleto').get(function(){
    return this.nombre + ' ' + this.apellido;
});

UsuarioSchema.set('toJSON', {getters: true, virtuals: true});
```

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.

```
UsuarioSchema.virtual('nombreCompleto')
.get(function(){
    return this.nombre + ' ' + this.apellido;
}).set(function(nombreCompleto){
    var nombreDividido = nombreCompleto.split(' ');
    this.nombre = nombreDividido[0] || '';
    this.apellido = nombreDividido[1] || '';
});
```

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 UsuarioSchmea para realizar esto

```
var UsuarioSchema = new Schema({
  ...
  usuario: {
    type: String,
    trim: true,
    unique: true
  },
  ...
});
```

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

```
var UsuarioSchema = new Schema({
  ...
  email: {
    type: String,
    index: true
  },
  ...
});
```

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.

```
const mongoose = require('../bin/mongodb');

var Schema = mongoose.Schema;

var autorSchema = new Schema({
    nombre: String,
    biografia: String,
    fecha_de_nacimiento: Date,
    nacionalidad: String
});

module.exports = mongoose.model('Autor', autorSchema);
```

supongamos un modelo sencillo para libro de la siguiente manera:

```
const mongoose = require('../bin/mongodb');

var Schema = mongoose.Schema;
var Autor = mongoose.model('Autor');

var libroSchema = new Schema({
    titulo: String
    paginas: Number,
    isbn: String,
    autor: { type: Schema.ObjectId, ref: "Autor" }
});

module.exports = mongoose.model("Libro", libroSchema);
```

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:

```
app.get("/libros", function(req, res) {
  Libro.find({}, function(err, libros) {
    res.status(200).send(libros)
  });
});
```

Luego aplicamos método populate dentro del callback de libros

```
app.get("/libros", function(req, res) {
  Libro.find({}, function(err, libros) {
    Autor.populate(libros, {path: "autor"},function(err, libros){
      res.status(200).send(libros);
    });
  });
});
```

Luego aplicamos método populate dentro del callback de libros

```
app.get("/libros", function(req, res) {
  Libro.find({}, function(err, libros) {
    Autor.populate(libros, {path: "autor"},function(err, libros){
      res.status(200).send(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:

```
[{  
    "_id": "547db17cbe9956a0000001",  
    "__v": 0  
    "titulo": "Juego de Tronos",  
    "paginas": 150,  
    "isbn": "0-553-57340-4",  
    "autor": {  
        "_id": "547db17cbe9958b0000001",  
        "__v": 0,  
        "nombre": "George R. R. Martin",  
        "biografia": "American novelist...",  
        "fecha_de_nacimiento": "1948-09-20T00:00:00.000Z",  
        "nacionalidad": "USA"  
    }  
},  
]
```

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

```
app.post("/libros", function(req, res) {  
    Autor.findOne({'Nombre':'Pablo'}, function(err, autor) {  
  
        Libros.create({  
            titulo: "Test"  
            paginas: 5,  
            isbn: "123456",  
            autor: autor._id  
        }, function (err, result) {  
            if (err)  
                next(err);  
            //next('route');  
            else  
                res.status(200).json({status: "success", message: "Libro added successfully!!!", data: result});  
        });  
    });  
});
```

Utilizando async / await

```
var productModel = require("../models/productsModel")
var categoriasModel = require("../models/categoriasModel")

module.exports = {
    getAll: async function(req, res, next) {
        var producto = await productModel.find({});
        var productoCompleto = await categoriasModel.populate(producto,{path:'categoria'});
        console.log(productoCompleto);
        res.status(200).json({status: "success", message: "ok", data: productoCompleto});
    },
    // Other methods like create, update, delete, etc.

    save: async function(req, res, next) {
        var result = await productModel.create({
            name: req.body.name,
            sku: req.body.sku,
            price: req.body.price,
            categoria:req.body.categoria
        });
        res.status(200).json({status: "success", message: "Product added successfully!!!", data: result});
    }
}
```

3. Subdocumentos

Subdocumentos

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

Definición Schema

```
const Schema = mongoose.Schema;
var childSchema = new Schema({ name: 'string' });
//Define a schema

const ProductSchema = new Schema({
  name: {
    type: String,
    trim: true,
    required: true,
  },
  sku: {
    type: String,
    trim: true,
    required: true
  },
  price: {
    type: Number,
    trim: true,
    required: true
  },
  categoria: {type:Schema.ObjectId, ref:"categorias"},
  relacionados:[childSchema]
});
module.exports = mongoose.model('products', ProductSchema);
```

Insertar datos

```
save: async function(req, res, next) {
  var product = new productModel({
    name: req.body.name,
    sku: req.body.sku,
    price: req.body.price,
    categoria: req.body.categoria
  });
  console.log(req.body.relacionados)
  product.relacionados.push(req.body.relacionados)
  var result = await product.save()

  res.status(200).json({status: "success", message: "Product added successfully!!!", data: result});
}
```

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

```
{
  "_id": "5d8049528f038622202b0f4f",
  "name": "Test1",
  "sku": "123132",
  "price": 2000,
  "categoria": {
    "_id": "5d80488c30cd71ec5f53549b",
    "name": "Celulares"
  },
  "relacionados": [
    {
      "_id": "5d8049528f038622202b0f50",
      "name": "rel1"
    }
  ],
  "__v": 0
}
```

Queries

```
Model.deleteMany()  
Model.deleteOne()  
Model.find()  
Model.findById()  
Model.findByIdAndDelete()  
Model.findByIdAndRemove()  
Model.findByIdAndUpdate()  
Model.findOne()  
Model.findOneAndDelete()  
Model.findOneAndRemove()  
Model.findOneAndReplace()  
Model.findOneAndUpdate()  
Model.replaceOne()  
Model.updateMany()  
Model.updateOne()
```

findByIdAndUpdate

```
update: async function(req, res, next) {  
    var data = await productModel.findByIdAndUpdate(req.params.id, { $set: { name: req.body.name }})  
    res.status(200).json({status: "success", message: "Product added successfully!!!", data: data});  
},
```

Ver todos los queries

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

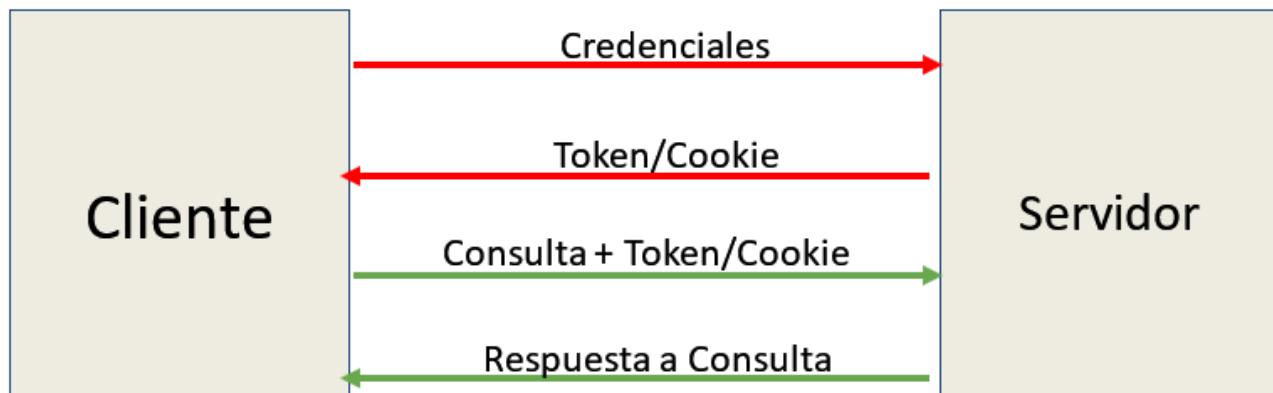
Validations

JWT

Autenticación y Autorización

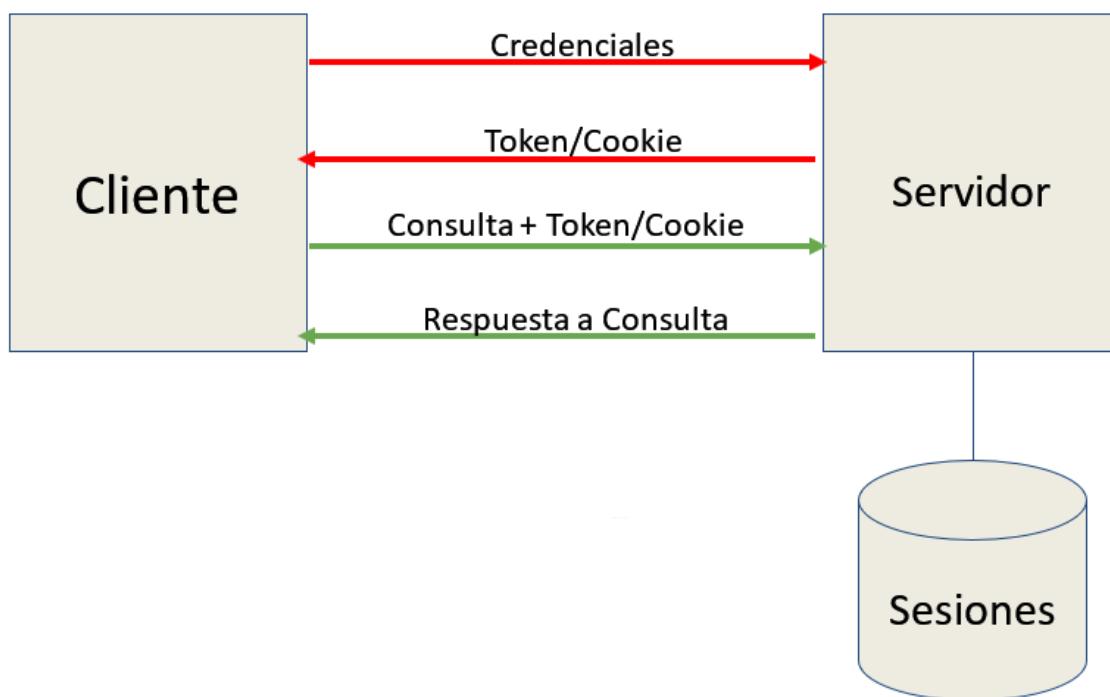
Definición de conceptos

- **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



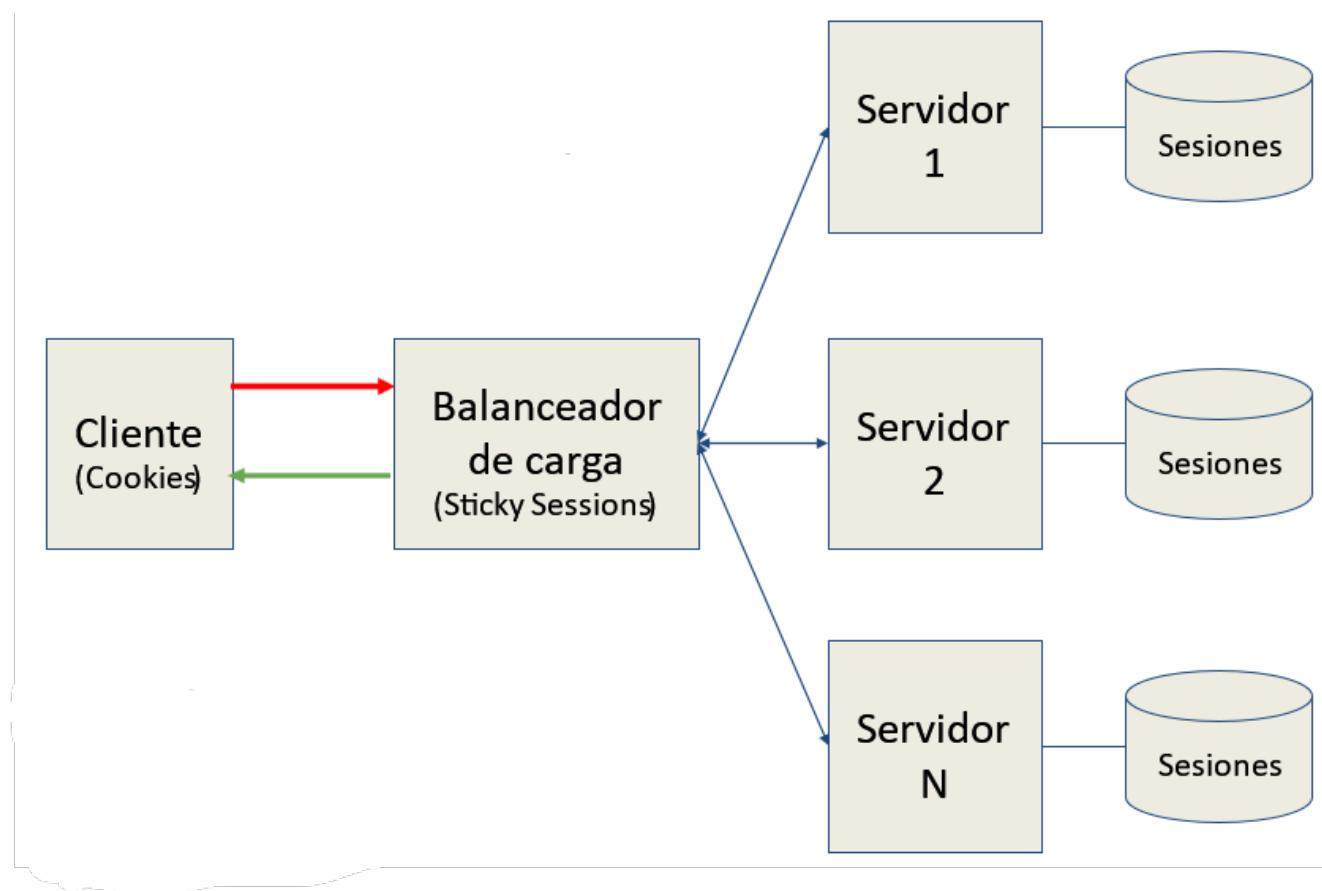
Ventajas

- 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



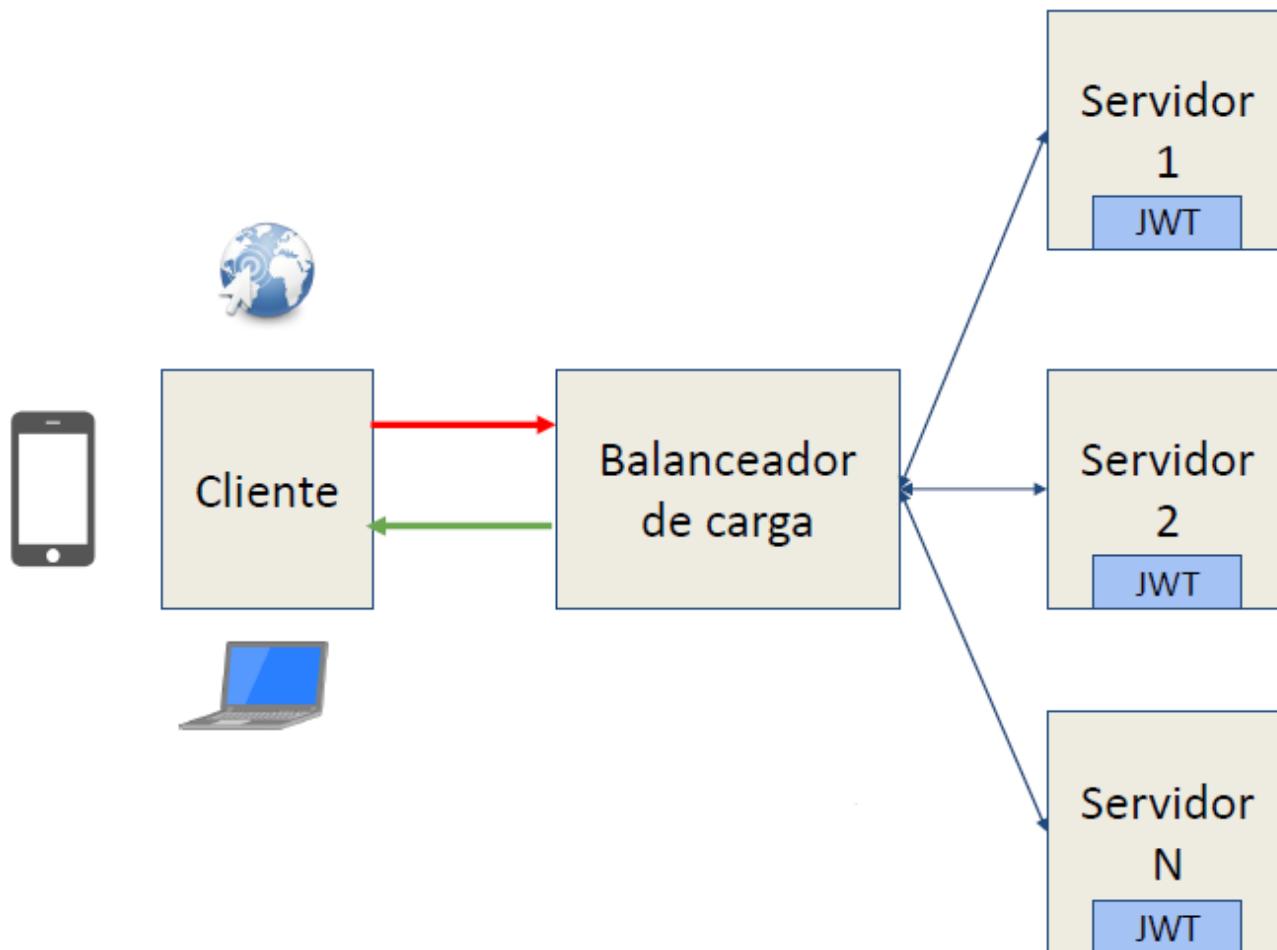
Ventajas

- 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 autocontenido”, es más eficiente en arquitecturas distribuidas, además que permite independencia en las plataformas cliente.

JSON WEB TOKEN

¿Qué es JWT?

JSON Web Token (**JWT**) es un **standard abierto** (RFC 7519) que **define** una forma **compacta** y **autocontenido** 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:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

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

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYLOAD: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022,  
  "dat":{  
    "user_role": "admin",  
    "user_email": "user@user.com"  
  }  
}
```

```
"dat":{  
  "user_role": "admin",  
  "user_email": "user@user.com"  
}  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
) secret base64 encoded
```

```
eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJzdWliOilxMjM0NTY3ODkwliwibmFtZSI6Ikpvag4  
gRG9lliwiaWF0IjoxNTE2MjM5MDIyLCJkYXQiOnsidXNlcI9yb2xlljoiYWRtaW4iLCJ1c2VyX2VtY  
WIsljoidXNlcIckB1c2VyLmNvbSJ9fQ.UVPhWil_QqdjOvv7vpxdfgRjVW8kBeLxUJ6jc3aEYIU
```

Librerías

JWT / Librerías

- .NET
- Python
- **NodeJs**
- Java
- **JavaScript**
- Perl
- Ruby
- Elixir
- Erlang
- Go
- Groovy
- Haskell
- Haxe
- Rust
- Lua
- Scala
- D
- Cloujure
- Objective-C
- Swift
- C
- C++
- kdb+/Q
- Delphi
- **PHP**
- Crystal
- 1C
- PostgreSQL

Instalación NodeJs

```
npm install jsonwebtoken
```

Uso básico

```
var jwt = require('jsonwebtoken');

var token = jwt.sign({ foo: 'bar' }, 'clave_secreta');
```

NodeJs - JWT / Crear nuevo token

```
var jwt = require('jsonwebtoken');

var token = jwt.sign({
    exp: Math.floor(Date.now() / 1000) + (60 * 60),
    data: 'foobar'
}, 'clave_secreta');
```

- 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

```
jwt.verify(token, 'clave_secreta', function(err,
decoded) {
    console.log(decoded.data) // foobar
});
```

```
try {
    var decoded = jwt.verify(token, 'wrong-secret');
} catch(err) {
    // err
}
```

BCRYPT

JWT - Bcrypt

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

npm install bcrypt

```
D:\pwaapp>npm install bcrypt
> bcrypt@3.0.6 install D:\pwaapp\node_modules\bcrypt
> node-pre-gyp install --fallback-to-build

node-pre-gyp [WARN] Using needle for node-pre-gyp https download
[bcrypt] Success: "D:\pwaapp\node_modules\bcrypt\lib\binding\bcrypt_lib.node" is installed via remote
+ bcrypt@3.0.6
added 60 packages from 47 contributors and audited 304 packages in 49.749s
found 1 moderate severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details
```

Aplicación

App.js

```
var jwt = require('jsonwebtoken');

var usersRouter = require('./routes/users');
var productosRouter = require('./routes/productos');
var autenticacionRouter = require('./routes/autentication');

//Definicion de secretKey
app.set('secretKey', 'nodeRestApi') // jwt secret token

app.use('/users', usersRouter);
app.use('/autentication', autenticacionRouter);
app.use('/products', validateUser,productosRouter);

function validateUser(req, res, next) {
  jwt.verify(req.headers['x-access-token'], req.app.get('secretKey'), function(err, decoded) {
    if (err) {
      res.json({status:"error", message: err.message, data:null});
    }else{
      // add user id to request
      req.body.userId = decoded.id;
      next();
    }
  });
}
```

Activar Wind
Ve a Configuraci

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. Método **validateUser**

Route de autenticación

```
var express = require('express');
var router = express.Router();
var authentication = require("../controllers/authentication")

/* GET home page. */
router.post('/registrar', authentication.save);
router.post('/login', authentication.login);
module.exports = router;
```

Controller de autenticación

```
var express = require('express');
var router = express.Router();
const bcrypt = require('bcrypt');
var authenticationModel = require("../models/authenticationModel")
const jwt = require('jsonwebtoken');
```

Require de modulos

- bcrypt
- jwt
- authenticationModel

```
module.exports = {
  save: async function(req, res, next) {
    var data = await authenticationModel.create({ name: req.body.name, usuario: req.body.usuario, password: req.body.password });
    res.json({status: "success", message: "User added successfully!!!", data: data});
  },
};
```

Registro de usuarios. Aplica método create de mongoose

```
login: async function(req, res, next) {
  var usuario = await autenticationModel.findOne({usuario:req.body.usuario});
  if (usuario) {
    if(bcrypt.compareSync(req.body.password, usuario.password)) {
      const token = jwt.sign({id: usuario._id}, req.app.get('secretKey'), { expiresIn: '1h' });
      console.log(token,usuario)
      res.json({status:"success", message: "user found!!!", data:{user: usuario, token:token}});
    }else{
      res.json({status:"error", message: "Invalid user/password!!!", data:null});
    }
  }else{
    res.json({status:"not_found", message: "user not found!!!", data:null});
  }
}
```

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

```
const mongoose = require('../bin/mongodb');
const bcrypt = require('bcrypt');
var UsuariosSchema = mongoose.Schema({
  name:String,
  usuario:{
    type: String,
    required: true
  },
  password:{
    type: String,
    trim: true,
    required: true
  }
})
UsuariosSchema.pre('save', function(next){
  this.password = bcrypt.hashSync(this.password, 10);
  next();
});
module.exports = mongoose.model('users', UsuariosSchema)]
```

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:

```
var UsuariosSchema = mongoose.Schema({  
    name: String,  
});
```

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

```
var UsuariosSchema = mongoose.Schema({  
    name: String,  
    usuario:{  
        type: String,  
        required: [true,"El campo usuario es obligatorio"],  
        unique: true  
    },  
});
```

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)

```
password:{  
    type: String,  
    trim: true,  
    required: [true,"El password es obligatorio"],  
    minlength: [6,"El password debe tener al menos 6 caracteres"],  
    maxlength: [8,"El password debe tener como máximo 8 caracteres"]  
},
```

En el campo password 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 maxlenlength sigue la misma regla

```
phone: {  
    type: String,  
    validate: {  
        validator: function(v) {  
            return /\d{2}-\d{4}-\d{4}/.test(v);  
        },  
        message: '{VALUE} no es un teléfono válido!'  
    },  
    required: [true, 'El campo teléfono es obligatorio']  
}
```

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

```
var producto = await productModel.find({})  
.sort({ price: 1 })
```

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

```
var producto = await productModel.find({})
    .select({ price: 1, name: 1})
```

Aplica el `select` de los atributos especificados (en el ejemplo `Price` y `name`), de esta forma solo se retornaran dichos datos.

```
var producto = await productModel.find({})
    .where('name').equals('Heladera');
```

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:

```
var mongoose = require('mongoose');
var mongoosePaginate = require('mongoose-paginate-v2');
mongoose.connect('mongodb://localhost/startup', { useNewUrlParser: true }, function(error){
  if(error){
    throw error;
  }else{
    console.log('Conectado a MongoDB');
  }
});
mongoosePaginate.paginate.options = {
  lean: true,
  limit: 1
};
mongoose.mongoosePaginate = mongoosePaginate
module.exports = mongoose;
```

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)

```
const mongoose = require('../bin/mongodb');
const Schema = mongoose.Schema;
var childSchema = new Schema({ name: 'string' });
//Define a schema
const ProductSchema = new Schema({
  name: {
    type: String,
    trim: true,
    required: true,
  },
  sku: {
    type: String,
    trim: true,
    required: true
  },
  price: {
    type: Number,
    trim: true,
    required: true
  },
  categoria: {type:Schema.ObjectId, ref:"categorias"},
  relacionados:[childSchema]
});
ProductSchema.plugin(mongoose.mongoosePaginate);
module.exports = mongoose.model('products', ProductSchema);
```

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

```
getAll: async function(req, res, next) {
  try{
    var producto = await productModel.paginate({
      },
      {
        populate: 'categoria',page:req.query.page
      })
    res.status(200).json({status: "success", message: "ok", data: producto});
  }catch(err){
    next(err);
  }
},
```

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

```
var query    = {};
var options = {
  select:  'title date author',
  sort:    { date: -1 },
  populate: 'author',
  lean:     true,
  offset:   20,
  limit:    10
};

Book.paginate(query, options).then(function(result) {
  // ...
});
```

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:

```
SECRET_KEY=nodeRestApi
MONGO_DB_HOST=localhost
MONGO_DB_DB=startup
MONGO_DB_LIMIT=1
MYSQL_HOST=localhost
MYSQL_DB=pwa
MYSQL_USER=root
MYSQL_PASSWORD=
```

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

```
//Definicion de secretKey
app.set('secretKey', process.env.SECRET_KEY); // jwt secret token
```

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

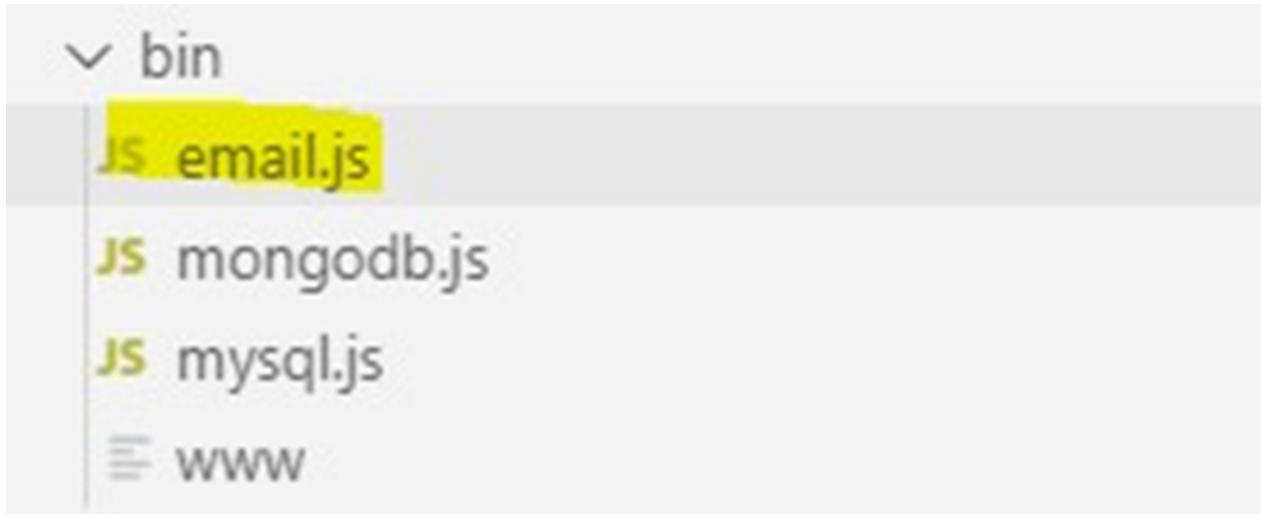
Nodemailer

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

Para instalar ejecutamos:

Npm install nodemailer

Luego creamos el archivo email.js dentro del directorio bin



Bin/email.js

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

let transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: process.env.EMAIL_USER, // generated ethereal user
    pass: process.env.EMAIL_PASS // generated ethereal password
  },
  tls: {
    rejectUnauthorized: false
  }
});

module.exports = transporter;
```

En caso de configurar una cuenta de otro cliente distinto a

Gmail, se debe hacer de la siguiente forma

Bin/email.js

```
let transporter = nodemailer.createTransport({  
    host: 'smtp.ethereal.email',  
    port: 587,  
    secure: false, // true for 465, false for other ports  
    auth: {  
        user: testAccount.user, // generated ethereal user  
        pass: testAccount.pass // generated ethereal password  
    }  
});
```

controller/autentication.js

```
var transporter = require("../bin/email")  
const jwt = require('jsonwebtoken');  
  
module.exports = {  
    save: async function(req, res, next) {  
        try{  
            var data = await autenticationModel.create({ name: req.body.name, usuario: req.body.usuario, email: req.body.email, password: req.body.password });  
            let info = await transporter.sendMail({  
                from: process.env.EMAIL_USER, // sender address  
                to: req.body.email, // list of receivers  
                subject: 'Bienvenido '+req.body.name, // Subject line  
                text: 'Bienvenido a este sitio', // plain text body  
                html: '<b>Bienvenido a este sitio</b>' // html body  
            });  
            res.json({status: "success", message: "User added successfully!!!", data: data});  
        }catch(err){  
            console.log(err)  
            next(err);  
        }  
    },  
    . . . . .  
};
```

Para ver mas acerca de nodemailer:

<https://nodemailer.com/>