ROLLINGCODE SCHOOL

Proyecto Backend

Proyecto Backend

1. Crear carpeta del proyecto y ejecutar desde consola **npm init** para inicializar el proyecto.
2. Ejecutar **npm install express --save** para instalar express que servirá para montar el servidor.
3. Crear un folder en la carpeta del proyecto con el nombre **server** y dentro crear un archivo js llamado **server.js**
4. En el archivo **server.js** escribir el siguiente código extraído de la documentación de express:
5. En la linea del puerto app.listen(3000) agrego un callback con un console.log que diga “Escuchando puerto 3000”
6. Para probar y ver el mensaje del servidor ejecuto en consola: **node server/server**
7. Como usaremos JSON vamos a cambiar en la petición *app.get* el **res.send** por **res.json()**
8. Modificamos la petición get y creamos el resto de las peticiones:

```
app.get('/usuarios', function (req, res) {
  res.json('GET usuarios')
}) app.post('/usuarios', function
(req, res) {
  res.json('POST usuarios')
}) app.put('/usuarios/:id', function
(req, res) {
  res.json('PUT usuarios')
}) app.delete('/usuarios/:id', function
(req, res) {
  res.json('DELETE usuarios')
})
```

9. Abrimos Postman para probar las peticiones con la ruta **http://localhost:3000/usuarios**. Recordar que en el caso de Put y Delete debemos agregar un id, así que al probarlas deberíamos hacerlo como el siguiente ejemplo **http://localhost:3000/usuarios/333**

10. Como necesitaremos enviar información al servidor para probar, activaremos en la solapa **body** de Postman la opción **x-www-form** y esto nos permitirá agregar un key que sería el nombre de la clave y un value que sería el contenido de dicha clave:

11. Para poder procesar la información que enviemos por body instalaremos el paquete body-parser: **npm install body-parser --save**

12. Importamos el paquete en server.js: **const bodyParser = require('body-parser')**

13. Agregamos debajo de la línea **const app = express()** lo siguiente extraído de la documentación del paquete:

14. Para probar que funcione modificaremos la petición post:
15. Probamos el cambio en Postman:

16. Podemos probar una validación desde la petición post, si el usuario no es enviado al servidor que devuelva un mensaje:

17. Probamos con Postman:

18. Crear archivo de configuración global:

    a. Dentro de la carpeta server crear una nueva llamada config

    b. Dentro de la carpeta config crear el archivo config.js

    c. Lo primero a configurar será el puerto usando **process.env,** luego seguiremos agregando más configuraciones:

19. En server.js requerimos el archivo config.js al inicio: **require('./config/config')**

20. Modificamos la línea de server.js que está escuchando el puerto 3000 para que tome el parámetro definido en el archivo de configuración:

21. Crear el archivo .gitignore en la raiz del proyecto y excluir node_modules/

22. Modificar el archivo package.json en scripts para que quede así:

23. Hacer un respaldo en github

24. Subir a Heroku creando una nueva aplicación:

    a. git push heroku master

    b. ***heroku open*** abrirá el navegador para ver que el server está subido correctamente.

25. Usar la ruta de Heroku para hacer pruebas en Postman agregando al final de la ruta ***/usuarios***

26. Configurar environment en Postman:

a. Click en la opción del margen superior derecho para acceder a crear un nuevo Environment

b. Si tienes la versión nueva ir a Environments en el menú de la izquierda y hacer click en el signo +

c. Coloca el nombre Desarrollo, variable:url, initial value: http://localhost:3000

d. Crea otro nuevo Environment con el nombre Producción, variable: url, initial value: la dirección de la app de heroku que creaste donde subiste el servidor sin ***/usuarios*** al final

27. Cuando quieras usar algún Environment debes elegirlo en el margen superior derecho y en la ruta de postman donde se hacen las pruebas debes escribir

{{url}}/usuarios

28. Instalamos mongoose para hacer conexión a la base de datos ***npm install mongoose --save***

29. En server.js debajo de la importación de express importamos mongoose: ***const mongoose = require('mongoose')***

30. Para establecer la conexión agregamos a server.js, antes de la linea que escucha el puerto, lo siguiente donde especificamos el puerto de mongoDB (27017) y la base de datos (test):

Documentación de Mongoose: https://mongoosejs.com/

31. En la carpeta server creo una nueva carpeta llamada ***rutas***. En esta carpeta creo el archivo ***usuario.js***

32. El archivo ***usuario.js*** contendrá las rutas o peticiones que habíamos configurado en server.js, por lo que ya no serán necesarias allí. Archivo usuario.js:

33. Debemos importar en server.js el archivo usuario.js debajo de la linea *app.use(bodyParser.json())* de la siguiente forma:

app.use(require('./rutas/usuario'))

34. Crear un nuevo folder en server llamado **modelos.**

35. Crear un nuevo archivo **usuario.js** dentro de modelos.

36. Importar moongose.

37. Obtener esquema:

    a. *let Schema = mongoose.Schema*

38. Definir el esquema:

    a. *let usuarioSchema = new Schema()*

39. Definir las reglas del esquema:

```
let usuarioSchema = new Schema({
  nombre: {
    type: String,
    required: [true, "El nombre es necesario"],
  },
  email: {
    type: String,
    required: [true, "El correo es necesario"],
  },
  password: {
    type: String,
    required: true,
  },
  img: {
    type: String,
    required: false,
  },
  role: {
    type: String,
    default: "USER_ROLE",
  },
  estado: {
    type: Boolean,
    default: true,
  },
});
```

40. Exportar el modelo al final:

```
module.exports = mongoose.model("Usuario", usuarioSchema);
```

41. Importar el modelo de usuario en el archivo usuario.js de rutas: `const Usuario = require('./modelos/usuario')`

Probar el modelo con la ruta del post:

```
let body = req.body;


  let usuario = new Usuario({
    nombre: body.nombre,
    email: body.email,
    password: body.password,
    role: body.role,
  });


  res.json({
    usuario,
  });
```

Grabar en la base de datos desde post, para eso cambiamos el código por:

```
  let usuario = new Usuario({
    nombre: body.nombre,
    email: body.email,
    password: body.password,
    role: body.role,
  });


  usuario.save((err, usuarioDB) => {
    if (err) {
      return res.status(400).json({
        ok: false,
```

```
        err,
      });
    }


    res.json({
      ok: true,
      usuario: usuarioDB,
    });
  });
```

Para que el email ingresado sea único, agregar en el modelo de usuario en el valor email la propiedad *unique:true*

```
  email: {
    type: String,
    required: [true, "El correo es necesario"],
    unique:true
  },
```

42. Descargar el paquete npm mongoose-unique-validator:

    a. npm i mongoose-unique-validator --save

43. Importarlo en el modelo de usuario:

```
const uniqueValidator = require('mongoose-unique-validator')
```

44. Antes de la linea del exports escribir:

```
usuarioSchema.plugin(uniqueValidator,{
    message: '{PATH} debe ser único'
})
```

45. De esta forma valido el email y mejoro el mensaje de error.

46. **Validar Roles**:

    a. En el modelo de usuario crear una variable debajo de las importaciones:

```
let rolesValidos={
    values:['ADMIN_ROLE', 'USER_ROLE'],
    message:'{VALUE} no es un rol válido'
}
```

b. En el listado del modelo de usuario buscar role y agregar ***enum:rolesValidos***

c. De esta forma cuando se mande un role que no está definido en rolesValidos arrojará un error y no guardará el registro.

47. **Encriptar contraseña**

a. Instalar bcrypt: **npm install bcrypt --save**

b. En la ruta de usuario importar bcrypt: ***const bcrypt = require('bcrypt')***

c. Ir a la ruta con el método post y en el password modificar para que quede de esta forma: `password: bcrypt.hashSync(body.password, 10),`

48. No mostrar la contraseña cuando se haga la petición:

a. Ir al modelo de usuario y agregar antes del module.exports:

```
usuarioSchema.methods.toJSON=function(){
    let user=this;
    let userObject=user.toObject();
    delete userObject.password;
    return userObject
}
```

49. **PUT: Actualizar info:**

a. En las rutas de usuario modificar put:

```
let body = req.body;

usuario.findByIdAndUpdate(id, body, {new:true},(err,
usuarioDB)=>
```

```
{
  if(err){
    return res.status(400).json({
      ok:false,
      err
    })
  }
  res.json({
    ok:true,
    usuario:usuarioDB
  })
})
```

50. **Validaciones adicionales:**
    a. Instalar underscore: ***npm install underscore --save***

    b. En las rutas de usuario importarlo debajo de bcrypt:

    const _=require('underscore')

    c. Modificar el método put para agregar los campos que se podrán actualizar con underscore y runValidators al método findByIdAndUpdate:

```
  let body = _.pick(req.body,
['nombre','email','img','role','estado']);


  usuario.findByIdAndUpdate(id, body, { new: true,
runValidators:true }, (err, usuarioDB) => {
    if (err) {
      return res.status(400).json({
        ok: false,
        err,
      });
    }
```

```
        res.json({
          ok: true,
          usuario: usuarioDB,
        });
      });
```

51. GET: Obtener datos:

a. Para traer datos modificar en rutas el método get. Esto trae todos los registros:

```
Usuario.find({})
.exec((err, usuarios)=>{
 if(err){
  return res.status(400).json({
   ok:false,
   err
  })
 }
 res.json({
  ok:true,
  usuarios
 })
})
```

b. Para limitar la cantidad de registros a mostrar agregar debajo de find({}) ***.limit(5)*** y esto limitará a 5 registros la lista.

c. Para saltarse los primeros 5 registros agregar debajo de find({}) ***.skip(5)***

```
app.get("/usuarios", function (req, res) {
 Usuario.find({})
  .limit(5)
  .skip(5)
  .exec((err, usuarios) => {
   if (err) {
```

```
        return res.status(400).json({
          ok: false,
          err,
        });
      }
      res.json({
        ok: true,
        usuarios,
      });
    });
});
```

d. Para manejar paginación hacemos uso de ***req.query*** donde se alojan parámetros opcionales. Antes de la función *Usuario.find* declaramos dos variables y las inicializamos:

```
let desde = req.query.desde || 0;
desde = Number(desde);


let limite = req.query.limite || 5;
limite = Number(limite)
```

e. Con estas variables manejaremos la paginación indicando los valores o dejando los que están por defecto. Debemos modificar .skip y .limit con las variables respectivas:

```
let desde = req.query.desde || 0;
desde = Number(desde);
let limite = req.query.limite || 5;
limite = Number(limite)


  Usuario.find({})
    .limit(limite)
```

```
.skip(desde)
```

f. Para probar en Postman escribimos

{{url}}/usuarios?limite=10&desde=5

Que muestre 10 registros por consulta y desde el registro 5 en adelante.

g. Para retornar la cantidad de registros cuando hagamos la petición get hacemos la modificación siguiente luego de controlar el error:

```
Usuario.count({}, (err, conteo) => {
  res.json({
    ok: true,
    usuarios,
    cantidad: conteo,
  });
});
```

h. Si queremos mostrar solo ciertos campos al hacer la petición get tendremos que modificar la linea de Usuario.find de la siguiente forma:

```
Usuario.find({ }, "nombre email role estado")
```

Es un string y se separa cada campo por espacios vacíos.

52. DELETE: Borrando un usuario fisicamente

a. En rutas vamos a usuario.js y en la linea ***app.delete*** agregamos lo siguiente:

```
app.delete("/usuarios/:id", function (req, res) {
  let id = req.params.id;
  Usuario.findByIdAndRemove(id, (err, usuarioBorrado) => {
    if (err) {
      return res.status(400).json({
        ok: false,
        err,
```

```
    });
  }
  res.json({
    ok: true,
    usuario: usuarioBorrado,
  });
 });


});
```

b. Para manejar si se envía el id de un usuario borrado o que no existe podemos modificar la petición delete de la siguiente forma:

```
app.delete("/usuarios/:id", function (req, res) {
 let id = req.params.id;
 Usuario.findByIdAndRemove(id, (err, usuarioBorrado) => {
  if (err) {
   return res.status(400).json({
    ok: false,
    err,
   });
  }


  if (!usuarioBorrado) {
   return res.status(400).json({
    ok: false,
    err: {
     message: "Usuario no encontrado",
    },
   });
  }


  res.json({
```

```
      ok: true,
      usuario: usuarioBorrado,
    });
  });
});
```

53. DELETE: Borrar usuario solo cambiando su estado a false:

a. Debemos crear una variable para manejar el estado.

b. Usar ***Usuario.findByIdAndUpdate*** en vez de Usuario.findByIdAndRemove.

```
app.delete("/usuarios/:id", function (req, res) {
  let id = req.params.id;

  let estadoActualizado = {
    estado: false,
  };

  Usuario.findByIdAndUpdate( id, { new: true }, (err, estadoActualizado, usuarioBorrado) => {

    if (err) {
      return res.status(400).json({
        ok: false,
        err,
      });
    }
```

```
    if (!usuarioBorrado) {
      return res.status(400).json({
        ok: false,
        err: {
          message: "Usuario no encontrado",
        },
      });
    }

    res.json({
      ok: true,
      usuario: usuarioBorrado,
    });
  }
 );
});
```

c. Para que al hacer la petición get solo muestre los usuarios activos debemos agregar en ***find({ })*** la condición de que solo muestre usuarios con la propiedad estado en true. Esta misma modificación se hace en ***Usuario.count*** dentro de app.get:

```
Usuario.find({ estado: true }, "nombre email role estado")
 .limit(limite)
 .skip(desde)
 .exec((err, usuarios) => {
  if (err) {
   return res.status(400).json({
    ok: false,
```

```
            err,
          });
        }


        Usuario.count({ estado: true }, (err, conteo) => {
          res.json({
            ok: true,
            usuarios,
            cantidad: conteo,
          });
        });
      });
```

54. Instalando MongoDB Atlas:

a. Ver video de instalación y configuración de MongoDB Compass:

https://drive.google.com/file/d/16DM26RZ3OPYl0_c1UPYHJZJ4fw4rxWIp/vie w?usp=sharing

55. TOKENS:

a. JSON WEB TOKEN (jwt)

i. Header: Algoritmo y tipo de token

ii. Payload: info que está en el token

iii. Firma: si el token es válido

b. Se pueden hacer pruebas en jwt.io

56. En la carpeta de rutas creamos un archivo llamado ***login.js***

57. Dentro de este archivo importamos express y bcrypt

58. Agregamos

a. *const Usuario = require('../modelo/usuario')*
b. *const app = express()*

c. Creamos una ruta post

d. al final *module.exports = app*

```
const express = require("express");
const bcrypt = require("bcrypt");

const Usuario = require("../modelos/usuario");
const app = express();


app.post('/login',(req,res)=>{

   res.json({
       ok:true
   })
})

module.exports = app;
```

59. Crear archivo de rutas

a. En la carpeta rutas creo un archivo ***index.js***

b. Con el siguiente código:

```
const express = require("express");
const app = express();


app.use(require("./usuario"));
app.use(require("./login"));


module.exports = app;
```

c. En el archivo **server.js** reemplazo la ruta *app.use(require('./rutas/usuario'))* por *app.use(require('./rutas/index'))*

60. Manejando login de usuario:

a. En el archivo ***login.js*** trabajamos con el post:

```
app.post("/login", (req, res) => {
let body = req.body;


Usuario.findOne({ email: body.email }, (err, usuarioDB) =>
{
  //Si hay un error de conexión
```

```
  if (err) {
    return res.status(500).json({
      ok: false,
      err,
    });
  }


  //Si el usuario no existe
  if (!usuarioDB) {
    return res.status(400).json({
      ok: false,
      err: {
        message: "Usuario o contraseña incorrectos",
      },
    });
  }


  //Si el usuario existe chequeamos la contraseña
          if (!bcrypt.compareSync(body.password,
                  usuarioDB.password)) {
```

```
            return res.status(400).json({
                ok:false,
                err:{
                    message:'Usuario o contraseña incorrectos'
                }
            });
        }


        //Si todo está ok
        res.json({
            ok:true,
            usuario:usuarioDB,
            token:'123' //Provisorio el token de prueba
        })
    });


});
```

61. JSON WEB TOKEN

a. Instalar:

npm install jsonwebtoken --save

b. En ***login.js*** debajo de la importación de bcrypt importamos jwt `const jwt = require('jsonwebtoken')`

c. Antes de la linea ***res.json({ok:true...})*** Agregamos la generación del token.

d. Luego modificamos la respuesta ok agregando el valor del token.

e. El código de los puntos c, d:

```
    let token=jwt.sign({
        usuario:usuarioDB
    },'este-es-el-seed',{expiresIn:'48h'})


    //Si todo está ok
```

```
  res.json({
    ok: true,
    usuario: usuarioDB,
    token: token
  });
```

f. Tanto la fecha de expiración del token como la semilla o secret deberian estar en ***config.js***

g. Agregamos al archivo ***config.js*** lo siguiente:

```
//Vencimiento del Token
process.env.CADUCIDAD_TOKEN='48h';


//Semilla de Token
process.env.SEED=process.env.SEED || 'este-es-el-seed'
```

h. Realizamos los reemplazos correspondientes en ***login.js:***

```
  let token = jwt.sign(
    {
      usuario: usuarioDB,
    },
    process.env.SEED,
    { expiresIn: process.env.CADUCIDAD_TOKEN }
  );
```

i. Crear la variable de la semilla en Heroku: ***heroku config: set SEED="este-es-el-seed-produccion"***

62. Proteger Rutas

a. Usar headers para mandar los tokens

b. Crear carpeta **middlewares**

c. Dentro de la carpeta crear el archivo ***autenticacion.js***

d. Contenido de autenticacion.js:

```
let verificaToken = (req, res, next) => {
 let token = req.get("token");
 res.json({
   token: token,
 });
};
```

```
module.exports = {
 verificaToken,
};
```

e. Importo en ***rutas/usuario.js*** debajo de ***const Usuario=require('../models/usuario)*** lo siguiente:

const {verificaToken}=require('../middlewares/autenticacion');

f. En la linea de **app.get** agregamos **verificaToken** para que quede de la siguiente forma:

```
app.get("/usuarios",verificaToken, function (req, res) {
```

g. Si ejecutamos en Postman la peticion GET http://localhost:3000/usuarios solo veremos el token (para ello debemos generarlo primero con POST http://localhost:3000/login enviando por body el email y password de un usuario)

63. Comprobar Token válido

a. Importar en ***autenticacion.js*** jwt

b. Modificamos el código de la siguiente forma para validar el token:

```
let verificaToken = (req, res, next) => {
 let token = req.get("token");


 jwt.verify(token, process.env.SEED, (err, decoded) => {
   if (err) {
     return res.status(401).json({
       ok: false,
       err: {
         message: "Token no válido",
       }
     });
   }
   req.usuario = decoded.usuario;
   next();
 });
```

64. Obtener información del Payload en cualquier servicio

a. En el archivo ***usuario.js*** de rutas agregamos a cada petición get, post, put, delete la función ***verificaToken***

65. Middlewares: Verificar rol

a. Solo los usuarios con Rol de administrador pueden agregar datos o actualizarlos.

b. En ***autenticacion.js*** agregamos otro middleware:

```
let verificaAdmin_role=(req, res, next)=>{
  let usuario = req.usuario
  if(usuario.role==='ADMIN_ROLE'){
    next()
  }else{
    return res.json({
      ok:false,
      err:{
        message:'El usuario no es administrador'
      }
    })
  }
}
```

c. Exportar verificaAdmin_role junto con verificaToken:

```
module.exports = {
  verificaToken,
  verificaAdmin_role
};
```

d. Ir a ***usuario.js*** de las rutas e importar en:

```
const { verificaToken, verificaAdmin_role } = require("../middleware/autenticacion");
```

e. Modificamos post, put y delete en ***usuario.js*** de rutas agregando entre corchetes verificaToken y verificaAdmin_role: app.post("/usuarios", [verificaToken, verificaAdmin_role]... app.put("/usuarios/:id", [verificaToken, verificaAdmin_role]... app.delete("/usuarios/:id", [verificaToken, verificaAdmin_role]...

66. Modelo categorias

a. Crear en la carpeta modelos el archivo ***categoria.js***

b. El contenido es el siguiente:

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

```
const Schema= mongoose.Schema
let categoriaSchema= new Schema({
   descripcion:{
      type: String,
      unique:true,
      required:[true, 'La descripción es necesaria']
   },
   usuario:{
      type:Schema.Types.ObjectId,
      ref:'Usuario'
   }

})

module.exports=mongoose.model('Categoria', categoriaSchema)
```

67. CRUD y rutas de categorias

a. Crear en la carpeta rutas el archivo ***categoria.js***
b. Importamos el archivo categoria.js en index.js:
`app.use(require("./categoria"));`
c. Contenido:

```
const express = require("express");
let {
  verificaToken,
  verificaAdmin_role,
} = require("../middleware/autenticacion");

let app = express();

let Categoria = require("../modelos/categoria");
```

```
app.get("/categoria", verificaToken, (req, res) => {
  Categoria.find({})
    .sort("descripcion") //ordeno de a-z por descripcion
    .populate("usuario", "nombre email") //traigo los datos del usuario
    que creó la
categoria
    .exec((err, categorias) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err,
        });
      }
      res.json({
        ok: true,
        categorias,
      });
    });
});
```

```
//Método Get con id
app.get("/categoria/:id", verificaToken, (req, res) => {
  let id = req.params.id;
  Categoria.findById(id, (err, categoriaDB) => {
    if (err) {
      return res.status(500).json({
        ok: false,
        err,
      });
```

```
    }
    if (!categoriaDB) {
      return res.status(500).json({
        ok: false,
        err: {
          message: "La categoría no existe",
        },
      });
    }
    res.json({
      ok: true,
      categoria: categoriaDB,
    });
  });
});


app.post("/categoria", [verificaToken, verificaAdmin_role], (req, res) => {
  let body = req.body;
  let categoria = new Categoria({
    descripcion: body.descripcion,
    usuario: req.usuario._id,
  });
  categoria.save((err, categoriaDB) => {
    if (err) {
      return res.status(500).json({
        ok: false,
        err,
      });
    }
    if (!categoriaDB) {
      return res.status(400).json({
```

```
        ok: false,
        err,
```

```
      });
    }
    res.json({
      ok: true,
      categoria: categoriaDB,
    });
  });
});


app.put("/categoria/:id", [verificaToken, verificaAdmin_role], (req, res) =>
{
  let id = req.params.id;
  let body = req.body;
  let descCategoria = {
    descripcion: body.descripcion,
  };
  Categoria.findByIdAndUpdate(
    id,
    descCategoria,
    { new: true, runValidators: true },
    (err, categoriaDB) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err,
        });
      }
      if (!categoriaDB) {
        return res.status(400).json({
```

```
            ok: false,
            err,
          });
        }

        res.json({
          ok: true,
          categoria: categoriaDB,
        });
      }
    );
});

app.delete(
  "/categoria/:id",
  [verificaToken, verificaAdmin_role],
  (req, res) => {
    let id = req.params.id;
    Categoria.findByIdAndRemove(id, (err, categoriaBorrada) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err,
        });
      }

      if (!categoriaBorrada) {
        return res.status(400).json({
          ok: false,
          err: {
```

```
            message: "El id no existe",
          },
        });
      }


      res.json({
        ok: true,
        message: "Categoría borrada",
      });
    });
  }
);
module.exports = app;
```

68. Modelo y rutas: Productos

a. En la carpeta modelos creamos el archivo ***producto.js***
b. El contenido es el siguiente:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;


const productoSchema = new Schema({
  nombre: {
    type: String,
    required: [true, "El nombre es necesario"],
  },
  precioUni: {
    type: Number,
    required: [true, "El precio únitario es necesario"],
  },
  descripcion: {
    type: String,
```

```
    required: false,
  },
  disponible: {
    type: Boolean,
    required: true,
    default: true,
  },
  categoria: {
    type: Schema.Types.ObjectId,
    ref: "Categoria",
    required: true,
  },
  usuario: {
    type: Schema.Types.ObjectId,
    ref: "Usuario",
  },
});


module.exports = mongoose.model("Producto", productoSchema);
```

c. Creamos el archivo ***producto.js*** en la carpeta rutas:

```
const express = require("express");
const { verificaToken } = require("../middlewares/autenticacion");


const app = express();
let Producto = require("../modelos/producto");


//---Método GET


app.get("/producto", verificaToken, (req, res) => {
```

```
let desde = req.query.desde || 0;
desde = Number(desde);

let limite = req.query.limite || 5;
limite = Number(limite);
```

```
Producto.find({ disponible: true })
 .limit(limite) //limito registros a mostrar por página
 .skip(desde) //desde que registro comienzo a mostrar
 .sort("nombre") //ordeno la lista por nombre A-Z
 .populate("usuario", "nombre email") //traigo los datos segun el id de usuario
 .populate("categoria", "descripcion") //traigo los datos segun id de categoria
 .exec((err, productos) => {
  if (err) {
   return res.status(500).json({
    ok: false,
    err,
   });
  }

  Producto.count({ disponible: true }, (err, conteo) => {
   if (err) {
    return res.status(400).json({
     ok: false,
     err,
    });
   }
   res.json({
```

```
            ok: true,
            productos,
            cantidad: conteo,
          });
        });
      });
});


app.get("/producto/:id", verificaToken, (req, res) => {
  let id = req.params.id;
  Producto.findById(id)
    .populate("usuario", "nombre email")
    .populate("categoria", "descripcion")
    .exec((err, productoDB) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err: {
            message: "El Id no existe o es incorrecto",
          },
```

```
        });
      }


      res.json({
        ok: true,
        producto: productoDB,
      });
    });
});
```

```
//============================
//  Buscar producto por termino
//============================
app.get("/producto/buscar/:termino", verificaToken, (req, res) => {
  let termino = req.params.termino;

  let reGex = new RegExp(termino, "i");

  Producto.find({ nombre: reGex })
    .populate("categoria", "descripcion")
    .exec((err, producto) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err,
        });
      }
      res.json({
        ok: true,
        producto,
      });
    });
});

app.post("/producto", verificaToken, (req, res) => {
  let body = req.body;

  let producto = new Producto({
```

```
    usuario: req.usuario._id,
    nombre: body.nombre,
    precioUni: body.precioUni,
    descripcion: body.descripcion,
    disponible: body.disponible,
    categoria: body.categoria,
  });

  producto.save((err, productoDB) => {
    if (err) {
      return res.status(500).json({
        ok: false,
        err,
      });
    }

    res.status(201).json({
      ok: true,
      producto: productoDB,
    });
  });
});

app.put("/producto/:id", verificaToken, (req, res) => {
  let id = req.params.id;

  let body = req.body;

  Producto.findByIdAndUpdate(
```

```
    id,
    body,
    { new: true, runValidators: true },
    (err, productoDB) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err,
        });
      }
      if (!productoDB) {
        return res.status(400).json({
          ok: false,
          err: {
            message: "El id no existe",
          },
        });
      }

      res.json({
        ok: true,
        message: "Producto actualizado",
        producto: productoDB,
      });
    }
  );
});

app.delete("/producto/:id", verificaToken, (req, res) => {
  let id = req.params.id;
```

```
let disponibleActualizado = {
  disponible: false,
};

Producto.findByIdAndUpdate(
  id,
  disponibleActualizado,
  { new: true },
  (err, productoBorrado) => {
    if (err) {
      return res.status(400).json({
        ok: false,
        err,
      });
    }
    if (!productoBorrado) {
      return res.status(400).json({
        ok: false,
        message: "Producto no encontrado",
      });
    }

    res.json({
      ok: true,
      producto: productoBorrado,
    });
  }
);
});
```

```
module.exports = app;
```

d. Agregar a las rutas en ***index.js:***
app.use(require('./producto'));

69. El archivo ***config.js*** debe tener el siguiente contenido:

```
//========Puerto==========
process.env.PORT = process.env.PORT || 3005;


//===Definir entorno====
process.env.NODE_ENV = process.env.NODE_ENV || "dev";


//===Base de datos======
let urlDB;


if (process.env.NODE_ENV === "dev") {
  urlDB = "mongodb://localhost:27017/rolling";
} else {
  urlDB = process.env.MONGO_URI;
}


process.env.URLDB = urlDB;


//===Caducidad de Token=====
process.env.CADUCIDAD_TOKEN = "48h";


//====Semilla==========
process.env.SEED = process.env.SEED || "este_es_el_semilla";
```

70. Crear la variable **MONGO_URI** en heroku y asignarle la ruta de la base de datos de mongo Atlas.
71. Guardamos todos los cambios y actualizamos el repositorio de Git y subimos los cambios a Heroku.
72. Para poder hacer la integración con el Front hay que instalar el paquete cors:

```
npm install cors --save
```

b. Importarlo en el archivo ***server.js:***

```
const cors = require("cors");
```

c. Debajo de la línea ***const app = express();*** agregar ***app.use(cors());***

EXTRAS:

1. Si llegaramos a tener errores con la dependencia bcrypt:
   a. bcrypt_lib.node is not a valid Win32 application
      i. instalar bcryptjs: `npm i bcryptjs --save`
      ii. Reemplazar todos los *const bcrypt = require("bcrypt")* por *const bcrypt = require("bcryptjs")* en las rutas ***usuario.js*** y ***login.js***