1 Node.js y MongoDB

1.1 Sumario

- 1 Node.js
  - 1.1 Introducción a Node.js
    - 1.1.1 La ventaja que aporta el motor V8 de Google
    - 1.1.2 Node.js mantiene un único subproceso
    - 1.1.3 Ejecución asíncrona sin bloqueo
    - 1.1.4 ¿Qué diferencias tiene Node.js respecto a Apache u otros servidores web?
    - 1.1.5 ¿Cuándo usar Node?
- 2 Instalacion de Node.js y npm (node packet manager) en Debian
  - 2.1 Instalación de Node.js en Debian
  - 2.2 Comandos básicos en el uso de Node.js
  - 2.3 Actualización de NodeJS a la última versión en Debian
- 3 El sistema de módulos en NodeJs
- 4 El core de Node.js y los módulos
- 5 MongoDB
  - 5.1 Instalación de MongoDB en Debian 10
  - 5.2 El shell de MongoDB
  - 5.3 Crear usuario administrador de MongoDB
  - 5.4 Habilitar la autenticación en MongoDB
  - 5.5 Listado de bases de datos creadas
  - 5.6 Creación de usuario y base de datos en MongoDB
  - 5.7 Borrado de un usuario en MongoDB
  - 5.8 Otros comandos de consulta en MongoDB
  - 5.9 Insertando datos en MongoDB
  - 5.10 Consultas en MongoDB
  - 5.11 Actualizando datos en MongoDB
  - 5.12 Borrando datos en MongoDB
- 6 Creación de un servidor web en Nodejs
- 7 Aplicación y API REST con MongoDB y NodeJS
  - 7.1 Creación de base de datos y tablas en nuestro servidor de MongoDB
    - 7.1.1 Creación de la base de datos tienda y usuario tienda en MongoDB
    - 7.1.2 Creación de las colecciones en MongoDB
  - 7.2 Carpeta inicial del proyecto e inicialización con npm init
  - 7.3 Instalación de paquetes adicionales para la aplicación
  - 7.4 Inicializando el servidor
  - 7.5 Rearranque automático del servidor de Node.js en modo de desarrollo
  - 7.6 Ejecución permanente de un script de Node.js en producción
  - 7.7 Estructura de directorios de la aplicación
    - 7.7.1 La aplicación principal index.js
    - 7.7.2 El Modelo
    - 7.7.3 Las Rutas
    - 7.7.4 Los Controladores
    - 7.7.5 Prueba inicial de la ruta test
  - 7.8 Conectando la aplicación a la base de datos tienda en MongoDB
  - 7.9 Configurando el módulo bodyParser
  - 7.10 Implementación de las operaciones de la API REST
    - 7.10.1 CREATE
    - 7.10.2 READ
    - 7.10.3 UPDATE
    - 7.10.4 DELETE
  - 7.11 Desarrollo de frontend de nuesta aplicación con Node.js, MongoDB, Bootstrap y jQuery
    - 7.11.1 Instalación de motor de plantillas Pug para Node.js
    - 7.11.2 Configuración de Pug en la aplicación
    - 7.11.3 Configuración de nueva ruta de la aplicación
    - 7.11.4 Creación del fichero controlador de la aplicación
    - 7.11.5 Creación de una plantilla Pug de ejemplo
    - 7.11.6 Creación de una ruta para probar la plantilla de ejemplo

◊ 7.11.7 Creación de una plantilla esqueleto de la aplicación
◊ 7.11.8 Configuración de aplicación principal index.js para trabajar con jQuery, Bootstrap y favicon
· 7.11.8.1 Fichero de rutas de la aplicación
· 7.11.8.2 ALTAS
· 7.11.8.3 BAJAS
· 7.11.8.4 MODIFICACIONES
· 7.11.8.5 CONSULTAS
◊ 7.11.9 Ruta a la página de Inicio de la aplicación
◊ 7.11.10 Descarga de la aplicación completa
• 8 Buenas prácticas de rendimiento. Mejora de Express.js en entorno de producción

2 Node.js

2.1 Introducción a Node.js

Node.js es un entorno **JavaScript de lado de servidor** que utiliza un **modelo asíncrono** y **dirigido por eventos**.

Node usa el motor de JavaScript V8 de Google: una VM tremendamente rápida y de gran calidad escrita por gente como Lars Bak, uno de los mejores ingenieros del mundo especializados en VMs.

No olvidemos que V8 es actualizado constantemente y es uno de los intérpretes más rápidos que puedan existir en la actualidad para cualquier lenguaje dinámico. Además las capacidades de Node para I/O (Entrada/Salida) son realmente ligeras y potentes, dando al desarrollador la posibilidad de utilizar a tope la I/O del sistema. Node soporta protocolos TCP, DNS y HTTP.

Una de las cosas más importantes por las que la gente se confunde cuando se explica Node.js es entender qué es exactamente. ¿Es un lenguaje diferente? ¿es solo un framework por encima o es algo más?

Node.js definitivamente no es un nuevo lenguaje, y no es solo un framework de JavaScript. Puede ser considerado como un entorno de tiempo de ejecución para JavaScript construido sobre el motor V8 de Google.

Entonces, nos proporciona un contexto donde podemos escribir código JavaScript en cualquier plataforma donde se puede instalar Node.js. ¡En cualquier sitio!

Ahora un poco sobre su historia! En 2009, Ryan Dahl hizo una presentación en JSConf eso cambió JavaScript para siempre. Durante su presentación, presentó a Node.js La comunidad JavaScript. Después de una charla de aproximadamente 45 minutos, concluyó, recibiendo una gran ovación de la audiencia en el proceso. Se inspiró para escribir Node.js después de ver una barra de progreso de carga de archivos simple en Flickr, el sitio para compartir imágenes.

Al darse cuenta de que el sitio estaba realizando todo el proceso de manera incorrecta, él decidió que tenía que haber una mejor solución.

Ahora veamos las características de Node.js, que lo hace único de otros lenguajes de programación del lado del servidor.

2.1.1 La ventaja que aporta el motor V8 de Google

El motor V8 fue desarrollado por Google y fue de código abierto en 2008. JavaScript es un lenguaje interpretado y no será tan eficiente como un lenguaje compilado, ya que cada línea de código se interpreta una por una mientras el código se ejecuta.

El motor V8 trae un modelo eficiente aquí, donde el código de JavaScript se compilará en código de nivel de máquina y las ejecuciones sucederán en el código compilado en lugar de interpretar el JavaScript. Pero a pesar de que NodeJS está utilizando el motor V8, Joyent, que es la compañía que mantiene NodeJS, no siempre actualiza el motor V8 a las últimas versiones que Google lanza activamente. Esto ha llevado a la nueva variante llamada io.js.

2.1.2 Node.js mantiene un único subproceso

Tal vez se pregunte, ¿cómo ayuda un modelo de subproceso único?

Típicamente PHP, ASP.Los servidores NET, Ruby o basados ??en Java siguen un modelo donde cada cliente solicita resultados con la instanciación

de un nuevo hilo o incluso un proceso. Cuando se trata de Node.js, las solicitudes se ejecutan en el mismo hilo, incluso con recursos compartidos.

¿Cuál será la ventaja de usar ese modelo de hilo único?

Primero debemos entender el problema que Node.js intenta resolver. Intenta hacer procesamiento asíncrono en un solo hilo para proporcionar más rendimiento y escalabilidad para aplicaciones que supuestamente manejan demasiado tráfico web.

Imagine aplicaciones web que manejan millones de solicitudes concurrentes; si el servidor crea un nuevo hilo para manejar cada solicitud que llega, consumirá muchos recursos y terminaríamos intentando agregar más y más servidores para aumentar la escalabilidad de la aplicación. El modelo de procesamiento asíncrono de subproceso único, tiene su ventaja en el contexto anterior, y puede procesar muchas más solicitudes concurrentes con menos cantidad de recursos del lado del servidor. Sin embargo, hay un desventaja de este enfoque: el nodo (por defecto) no utilizará la cantidad de CPU cores disponibles en el servidor en el que se ejecuta, sin utilizar módulos adicionales como pm2.

2.1.3 Ejecución asíncrona sin bloqueo

Una de las características más poderosas de NodeJs es que está dirigido por eventos y asíncrono. Entonces, ¿cómo funciona un modelo asincrónico?

Imagina que tienes un bloque de código y en alguna enésima línea tiene una operación que requiere mucho tiempo.

Y qué sucede con las líneas que siguen a la enésima línea mientras se ejecuta este código? En modelos de programación síncrona, las líneas que siguen a la enésima línea tendrán que esperar hasta que se complete la operación en esa línea. Un modelo asincrónico maneja este caso de forma distinta. Para gestionar este escenario en un enfoque asincrónico, necesitamos segmentar el código que sigue a la enésima línea en dos secciones: la primera sección es dependiente del resultado de la operación en la enésima línea y la segunda sección es independiente del resultado.

Envolvemos el código dependiente en una función con el resultado de la operación como su parámetro y lo registramos como una función de retorno (callback) que se llamará cuando terminó con éxito la operación. Una vez la operación está completa, la función de devolución de llamada se activará con su resultado. Mientras tanto, podemos continuar ejecutando las líneas independientes del resultado sin esperar el resultado.

En este escenario, la ejecución nunca se bloquea para que se complete un proceso. Solo va con funciones de devolución de llamada registradas en cada finalización. En pocas palabras, usted asigna una función de devolución de llamada a una operación, y cuando NodeJS determina que ha finalizado la tarea, se activa el evento, y ejecutará su función de devolución de llamada en ese momento.

Para entenderlo veamos un ejemplo de asincronía en detalle:

```
console.log('Uno');
console.log('Dos');
setTimeout(function() {
   console.log('Tres');
}, 2000);
console.log('Cuatro');
console.log('Cinco');

# En una llamada típica síncrona el código anterior devolvería:

Uno
Dos
... (2 segundos de espera) ...
Tres
Cuatro
Cinco

# Sin embargo en una llamada asíncrona, el resultado sería:

Uno
Dos
Cuatro
Cinco
... (approx. 2 segundos de espera) ...
Tres

# La función que actualmente muestra "Tres" y que está asignada al TimeOut se denomina función de callback o retorno.
```

2.1.4 ¿Qué diferencias tiene Node.js respecto a Apache u otros servidores web?

- Apache crea un nuevo hilo por cada conexión cliente-servidor. Esto funciona bien para pocas conexiones, pero crear nuevos hilos tiene un coste, así como la realización de cambios de contexto. A partir de 400 conexiones simultáneas, el número de segundos para atender las peticiones crece considerablemente. Podemos decir que Apache funciona bien pero no es el mejor servidor para lograr máxima concurrencia (lograr tener el número mayor de conexiones abiertas posibles).
- Uno de los puntos fuertes de Node es su capacidad de mantener muchas conexiones abiertas y esperando. En Apache por ejemplo el parámetro **MaxClients por defecto es 256**. Este valor puede ser aumentado para servir contenido estático, sin embargo si se sirven aplicaciones web dinámicas en PHP u otro lenguaje es probable que al poner un valor alto el servidor se quede bloqueado ante muchas conexiones -esto dependerá del trabajo que la aplicación web de al servidor y de su capacidad hardware-.
- Una aplicación para Node se programa sobre un solo hilo. Si en la aplicación existe una operación bloqueante (entrada/salida por ejemplo), Node creará entonces otro hilo en segundo plano, pero no lo hará sistemáticamente por cada conexión como haría Apache.
- En teoría Node puede mantener tantas conexiones como número máximo de archivos descriptores (sockets) soportados por el sistema. En un sistema UNIX este límite puede rondar por las **65.000 conexiones**, un número muy alto. Sin embargo en la realidad la cifra depende de muchos factores, como la cantidad de información que esté la aplicación distribuyendo a los clientes.
- Una aplicación con actividad normal podría mantener **20.000-25.000** clientes a la vez sin haber apenas retardo en las respuestas. Un inconveniente de Node es que debido a su arquitectura (de usar sólo un hilo) también sólo podrá usar una CPU. Un método para usar múltiples núcleos sería iniciar múltiples instancias de Node en el servidor y poner un balanceador de carga delante de ellos.

Node.js no es el único servidor de este tipo, hay otros proyectos como Tornado (Python), Twisted(Python),Apache Mina(Java), Jetty(Java), etc.

Otra buena alternativa puede ser el lenguaje Erlang el cuál permite crear sistemas en tiempo real con alta escalabilidad y alta disponibilidad.

2.1.5 ¿Cuándo usar Node?

Node.js es muy adecuado para aplicaciones que se espera que vayan a gestionar una gran cantidad de conexiones concurrentes. Además, debe tenerse en cuenta que es más adecuado para aplicaciones donde cada solicitud entrante requiere muy poca CPU o ciclos. Esto significa que si tiene la intención de realizar tareas de computación intensivas en las solicitudes, terminará bloqueando el bucle de eventos, lo que afectará otras solicitudes que se soliciten al servidor web.

Node.js es muy adecuado para aplicaciones web en tiempo real, como salas de chat, herramientas de colaboración, juegos en línea, etc. Entonces para decidir si usar o no usar Node.js, debemos analizar el contexto de la aplicación en serio y descubrir si Node.js realmente se adapta al contexto de la aplicación.

3 Instalacion de Node.js y npm (node packet manager) en Debian

Veamos como realizar la instalación de Node.js y el gestor de paquetes de Node npm.

3.1 Instalación de Node.js en Debian

Para instalar la última versión de Nodejs haremos lo siguiente:

```
curl -sL https://deb.nodesource.com/setup_13.x | sudo bash -

# Instalamos a continuación nodejs y npm
apt-get install -y nodejs

# Chequeamos la versión instalada:
node -v

v13.5.0
```

3.2 Comandos básicos en el uso de Node.js

Desde la línea de comandos de node podemos teclear instrucciones para ejecutar aplicaciones o bien escribir una aplicación. En general lo que se suele hacer es crear un fichero .js con las instrucciones de la aplicación que va a ejecutar Node.

El entorno de trabajo al que se accede desde la línea de comandos se denomina REPL.

```
// Para acceder a la línea de comandos REPL de node:
node
```

```
// Accederemos a la consola de trabajo indicada con el símbolo >

// Para consultar ayuda de node:
> .help

// Para salir de node:
> .exit
// O bien CTRL+C CTRL+C

// Ejemplo de código de javascript en el entorno REPL de node:
>a=4;
4


// Ejemplo de pequeño programa en linea de comandos >:
> a=[1,2,3];
> a.forEach(function(valor){
..... console.log(valor);
..... });

// Producirá como resultado
1
2
3
```

Ejemplo de video con comandos de prueba en el entorno REPL de Node:

3.3 Actualización de NodeJS a la última versión en Debian

```
# Limpiamos la caché
npm cache clean -f

# Instalamos el módulo n
npm install -g n

# Descargamos la última versión disponible.
n latest

# Si quisiéramos descargar la versión estable
n stable

# Salimos de la shell y volvemos a entrar y comprobamos la versión
node -v

# Y nos devolverá algo como:
v13.9.0
```

4 El sistema de módulos en NodeJs

En un esfuerzo por hacer que el código sea lo más modular y reutilizable posible, Node utiliza un sistema de módulos que le permite organizar mejor su código. La idea básica es escribir un código que soluciona un problema y se exporta como un módulo.

Luego, cada vez que necesite usar ese código en otro lugar en su código, lo cargaremos y lo usaremos, ejemplo:

```
// ** archivo: sumas.js
module.exports = {
  sumar: function (param1, param2) {
    return param1 + param2;
  }
}

// ** archivo: pruebas.js

var mimodulo = require ('./sumas'); // nota: fijarse que no lleva la extensión .js en el archivo
var algo = 1;
var otracosa = 2;
```

```
var nuevoValor= mimodulo.sumar (algo, otracosa);
console.log (nuevoValor);
// => 3
```

Con este sistema, es fácil reutilizar la funcionalidad de un módulo (en este caso, el módulo sumas) en otros archivos.

Además, los archivos individuales de un módulo actúan como un espacio de nombres privado. Cualquier variable declarada y utilizada dentro del archivo del módulo es privado para ese módulo y no está expuesto a ningún código que use el módulo a través de require ().

Este sistema también se extiende infinitamente. Dentro de unos módulos módulos, se pueden requerir otro módulos y así sucesivamente.

5 El core de Node.js y los módulos

El núcleo de Node.js literalmente tiene cientos de módulos disponibles a la hora de escribir aplicaciones.

Entre ellos tenemos los siguientes:

- Events
- HTTP
- Filesystems
- Net
- Streams
- Timers

Para instalar los módulos lo haremos utilizando npm (Node Package Manager):

Por ejemplo:

```
npm install express

// Luego podremos utilizarlo en un fichero

var express = require ('express');
```

6 MongoDB

MongoDB es una base de datos escalable, de alto rendimiento, orientada a documentos y sin esquema predefinido.

MongoDB se aleja del clásico paradigma relacional. En este sentido, podemos llamarla una **base de datos ?NoSQL?**, que es un término que se acuñó hace unos años para indicar que no se va a emplear SQL, ampliamente extendido en las bases de datos comerciales como Oracle, MySQL, MariaDB, SQL Server, etc.

Se trata de un sistema de bases de datos de código abierto y completamente gratuito, si bien la empresa que lo creó ofrece servicios profesionales a empresas.

Y si no usa SQL, ¿qué utiliza en su lugar?

El sistema MongoDB está **orientado a documentos**.

Estos documentos se crean y consultan en **formato JSON** (Java Script Object Notation) y son fácilmente modificables y accesibles. Esto hace que sea una base de datos ideal para desarrollo ágil (agile development) y que permita a los desarrolladores hacer modificaciones rápidamente en su código.

Uno de los principales objetivos de los creadores de MongoDB era hacer una base de datos que permitiera a los desarrolladores interactuar con ella de la forma más sencilla posible.

¿Es mejor o peor que una base de datos SQL?

Un sistema de bases de datos relacional (SQL) almacena los datos en tablas con filas y campos, mientras que MongoDB lo hace en colecciones de documentos JSON (aunque internamente trabaja con BSON (Binary JSON).

Los desarrolladores han sacrificado algunas de las características que ofrecen las bases de datos relacionales (transacciones, normalización), para acercarse a las necesidades de los desarrolladores de aplicaciones (flexibilidad, escalabilidad).

Existen drivers de conexión a MongoDB para la mayoría de los lenguajes de programación, sistemas de balanceo de carga, posibilidad de escalar horizontalmente, y una capacidad ilimitada de almacenamiento.

El tipo de licencia es GNU APGL, y funciona en Windows, Linux, OS X y Solaris, con lo cual tampoco nos dará un quebradero de cabeza a la hora de comprar la licencia como otras bases de datos comerciales, pues es totalmente gratuita.

6.1 Instalación de MongoDB en Debian 10

```
# Nos pasamos a root
sudo su

# Importamos la clave GPG de MongoDB
apt update
apt -y install gnupg2

# Ejecutar la siguiente línea
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc | apt-key add -

# Añadimos el repositorio de MongoDB
echo "deb http://repo.mongodb.org/apt/debian buster/mongodb-org/4.2 main" | tee /etc/apt/sources.list.d/mongodb-org.list

# Instalamos la última versión estable:
apt update
apt install mongodb-org

# Para activar el servicio de MongoDB en el arranque:
systemctl enable --now mongod

# Podemos comprobar el estado de MongoDB con:
systemctl status mongod.service

# Obtendremos algo como:
root@dwes:/home/veiga# systemctl status mongod.service
? mongod.service - MongoDB Database Server
   Loaded: loaded (/lib/systemd/system/mongod.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2019-12-04 23:35:46 CET; 30s ago
     Docs: https://docs.mongodb.org/manual
 Main PID: 13262 (mongod)
   Memory: 105.0M
   CGroup: /system.slice/mongod.service
           ??13262 /usr/bin/mongod --config /etc/mongod.conf

Dec 04 23:35:46 dwes systemd[1]: Started MongoDB Database Server.


# Para ver el puerto en el que está escuchando:
ss -tunelp

# Y veremos que está escuchando en localhost (127.0.0.1) y en el puerto 27017

root@dwes:/home/veiga# ss -tunelp
Netid  State   Recv-Q  Send-Q   Local Address:Port    Peer Address:Port
udp    UNCONN  0       0              0.0.0.0:68           0.0.0.0:*     users:(("dhclient",pid=287,fd=7)) ino:10558 sk:1 <->
tcp    LISTEN  0       20           127.0.0.1:25           0.0.0.0:*     users:(("exim4",pid=797,fd=3)) ino:13855 sk:2 <->
tcp    LISTEN  0       128            0.0.0.0:443          0.0.0.0:*     users:(("nginx",pid=425,fd=9),("nginx",pid=424,fd=9)) ino:1
tcp    LISTEN  0       128          127.0.0.1:27017        0.0.0.0:*     users:(("mongod",pid=13262,fd=11)) uid:109 ino:68111 sk:4 <
tcp    LISTEN  0       80           127.0.0.1:3306         0.0.0.0:*     users:(("mysqld",pid=422,fd=22)) uid:107 ino:12319 sk:5 <->
tcp    LISTEN  0       128            0.0.0.0:80           0.0.0.0:*     users:(("nginx",pid=425,fd=6),("nginx",pid=424,fd=6)) ino:1
tcp    LISTEN  0       128            0.0.0.0:22           0.0.0.0:*     users:(("sshd",pid=534,fd=3)) ino:13324 sk:7 <->
tcp    LISTEN  0       128               [::]:443             [::]:*     users:(("nginx",pid=425,fd=8),("nginx",pid=424,fd=8)) ino:1
tcp    LISTEN  0       128               [::]:80              [::]:*     users:(("nginx",pid=425,fd=7),("nginx",pid=424,fd=7)) ino:1
tcp    LISTEN  0       128               [::]:22              [::]:*     users:(("sshd",pid=534,fd=4)) ino:13335 sk:a v6only:1 <->

# El puerto de escucha lo podríamos modificar editando el fichero /etc/mongod.conf:
nano /etc/mongod.conf

# En la sección de network interfaces modificaremos los datos que queramos:
```

```
# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1

# Una vez hechos los cambios reiniciamos con:
systemctl restart mongod.service
```

6.2 El shell de MongoDB

Para acceder a MongoDB desde el shell haremos:

```
mongo

# Para salir teclearemos:

exit
```

6.3 Crear usuario administrador de MongoDB

MongoDB por defecto no está configurado con ningún usuario administrador. Se recomienda por lo tanto crear un usuario que nos permita gestionar el resto de usuarios y bases de datos.

- El rol **admin vs root**: El rol **userAdminAnyDatabase** en MongoDB proporciona la capacidad de **crear usuarios** y **asignarles roles**, pero por si mismo **no permite al usuario hacer ninguna otra cosa**.
- El rol de **superusuario** en MongoDB es **root**.

El **usuario administrador** que hagamos en MongoDB se añadirá a la base de datos **admin** de MongoDB, de la siguiente forma:

```
# Primeramente nos conectamos a mongo
mongo

# Nos conectamos a la base de datos '''admin''' y creamos un usuario nuevo, por ejemplo con el nombre de '''admin''' o el que nosotr
use admin

switched to db admin

> db.createUser(
{
   user: "admin",
   pwd: "abc123.",
   roles: [ "root" ]
}
)

Successfully added user: { "user" : "username", "roles" : [ "root" ] }

# A partir de este momento el nuevo usuario podrá usarse para loguearse en la base de datos MongoDB con el siguiente comando:
mongo -u username -p password --authenticationDatabase admin mydatabase

# Pero para ello tenemos que habilitar la autenticación en mongoDB.
# Veáse en siguientes apartados como activar la autenticación.
```

6.4 Habilitar la autenticación en MongoDB

Para poder conectarnos con usuario y contraseña a mongoDB deberemos **habilitar la autenticación** o comprobar que está activada y para ello haremos lo siguiente:

```
# Como usuario root de Linux:
sudo su

# Para configurar y habilitar la autenticación en mongodb:
# Abrimos /etc/mongod.conf
nano /etc/mongod.conf

# Habilitamos la autenticación o comprobamos que estas 2 líneas están descomentadas:
```

```
security:
  authorization: enabled

# Reiniciamos el servicio:
systemctl restart mongod

# A partir de ahora ya nos podremos conectar con un usuario y contraseña a cualquier base de datos:
mongo -u username -p password --authenticationDatabase mydatabase

# Ejemplo de uso de usuario admin autenticándose con la base de datos admin:
mongo -u admin -p --authenticationDatabase admin

# Otra forma de conexión:
mongo direccionIP:puerto/basedatos
mongo localhost:27019/admin

# Conectándonos a un puerto distinto/base_de_datos, -usuario y autenticación:
mongo localhost:27018/pruebas -u pruebas --authenticationDatabase pruebas
```

6.5 Listado de bases de datos creadas

```
# Nos conectamos primeramente al servidor:
mongo --port 27017

# Si queremos ver las bases de datos que hay (ATENCION: Solamente mostrará aquellas bases de datos que no estén vacías):
> show databases;
admin   0.000GB
config  0.000GB
local   0.000GB

# Para usar una base de datos en particular, se hace con el comando use basedatos:
# Ejemplo:
use admin
```

6.6 Creación de usuario y base de datos en MongoDB

Para crear nuevos usuarios y bases de datos en el sistema:

```
# Primero conectaremos a mongoDB con el usuario admin
mongo -u admin -p abc123. --authenticationDatabase admin

# Nos conectaremos a la base de datos dónde queremos crear los nuevos usuarios, ejemplo base de datos pruebas:
> use pruebas
switched to db pruebas

# Según el manual de MongoDB, la función db.createUser() toma como parámetro un documento de tipo "user".
# Este documento (como todos en MongoDB) utiliza un formato estilo JSON que requiere la presencia de las claves "user", "pwd" y "rol
# Es posible indicar parámetros adicionales, los cuales son opcionales.
# Los roles predefinidos son "read", "readWrite", "dbAdmin", "dbOwner", "userAdmin", entre otros.
# Más info: https://docs.mongodb.com/manual/reference/method/db.createUser/
# Más info sobre roles de usuario: https://docs.mongodb.com/manual/reference/built-in-roles/
# Para crear un usuario entonces, es necesario invocar al método createUser() y pasar un documento "user" como parámetro:

# Crearemos a continuación un usuario con el mismo nombre del de la base de datos y con privilegio de propietario de la base de dato
> db.createUser(
... {
... user: "pruebas",
... pwd: "abc123.",
... roles: ["dbOwner"]
... }
... )
Successfully added user: { "user" : "pruebas", "roles" : [ "dbOwner" ] }

# En roles podemos poner "dbOwner" .
# Si queremos que sea un usuario normal con permisos de lectura/escritura pero no dbOwner, entonces le pondremos como role: "readWri

# Podemos consultar los usuarios de la base de datos con el comando "show users" o '''db.getUsers()''':

> show users
{
        "_id" : "pruebas.pruebas",
```

```
        "userId" : UUID("a96663ec-e4a6-4d03-afb4-0385ff66d789"),
        "user" : "pruebas",
        "db" : "pruebas",
        "roles" : [
                {
                        "role" : "dbOwner",
                        "db" : "pruebas"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}

# Si queremos que el usuario tenga privilegios en más de una base de datos en MongoDB:
# Por ejemplo con este comando estamos creando un usuario en la base de datos en la que nos encontramos pero con permisos de lectura

> db.createUser(
  {
    user: "pepe",
    pwd: "abc123",
    roles: [
             { role: "readWrite", db: "pruebas" },
             { role: "read", db:"nominas" }
           ]
  }
)

# Nos desconectamos de mongodb y nos conectamos con el usuario pruebas a la base de datos pruebas que acabamos de hacer:

# Para conectarnos a partir de este momento con ese usuario lo haremos indicando la base de datos con --authenticationDatabase prueb
mongo -u pruebas -p abc123. --authenticationDatabase pruebas

# O bien nos puede solicitar la contraseña al autenticarnos:
mongo -u pruebas --authenticationDatabase pruebas
```

6.7 Borrado de un usuario en MongoDB

```
# Nos conectaremos a la base de datos que queramos
> use pruebas

# Y una vez dentro ejecutaremos el comando: db.dropUser("usuario")
> db.dropUser("pepe")
```

6.8 Otros comandos de consulta en MongoDB

- Si queremos consultar las estadísticas de la base de datos:

```
# Si queremos ver estadísticas de la base de datos;
> db.stats()
{
        "db" : "pruebas",
        "collections" : 0,
        "views" : 0,
        "objects" : 0,
        "avgObjSize" : 0,
        "dataSize" : 0,
        "storageSize" : 0,
        "numExtents" : 0,
        "indexes" : 0,
        "indexSize" : 0,
        "scaleFactor" : 1,
        "fileSize" : 0,
        "fsUsedSize" : 0,
        "fsTotalSize" : 0,
        "ok" : 1
}

# Con el comando db.help() vemos la ayuda de mongodb:
> db.help()
```

```
DB methods:
        db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command [just calls db.runCommand(...)]
        db.aggregate([pipeline], {options}) - performs a collectionless aggregation on this database; returns a cursor
        db.auth(username, password)
        db.cloneDatabase(fromhost) - will only function with MongoDB 4.0 and below
        db.commandHelp(name) returns the help for the command
        db.copyDatabase(fromdb, todb, fromhost) - will only function with MongoDB 4.0 and below
        db.createCollection(name, {size: ..., capped: ..., max: ...})
        db.createUser(userDocument)
        db.createView(name, viewOn, [{$operator: {...}}, ...], {viewOptions})
        db.currentOp() displays currently executing operations in the db
        db.dropDatabase(writeConcern)
        db.dropUser(username)
        db.eval() - deprecated
..........

# Para comprobar que estamos en la base de datos correcta, lo hacemos con el comando db:
> db
pruebas
```

6.9 Insertando datos en MongoDB

Vamos a ver como podemos insertando datos en una colección de una base de datos. En este caso lo vamos hacer sobre la base de datos pruebas.

```
> use pruebas
switched to db pruebas

# Comprobamos que estamos en la base de datos pruebas:
> db
pruebas

# Ahora insertaremos 3 registros en una tabla llamada "personas", por supuesto en formato JSON.
# Utilizaremos el comando db.NombreTabla.save(ObjetoJSON).
# Con el comando .save() si no proporcionamos un id inserta un registro, si le proporcionamos un id, actualizará el registro con ese
# También tenemos la opción de utilizar el comando .insert(). Si le pasamos un id con este método, si el id ya existe devolverá un e

> db.personas.save({nombre:"Pepito Perez",localidad:"Santiago de Compostela",provincia:"A Coruña"})
WriteResult({ "nInserted" : 1 })

> db.personas.save({nombre:"Maria Yolanda Veiga",localidad:"Ourense",provincia:"Ourense"})
WriteResult({ "nInserted" : 1 })

> db.personas.save({nombre:"Jesus Fernandez",localidad:"Lugo",provincia:"Lugo"})
WriteResult({ "nInserted" : 1 })

> db.personas.insert({nombre:"Rafa",localidad:"Ourense",provincia:"Ourense"})
WriteResult({ "nInserted" : 1 })
```

6.10 Consultas en MongoDB

```
# Si queremos ver las colecciones que hay en la base de datos pruebas
# Nos conectamos a la base de datos pruebas:

> use pruebas
switched to db pruebas

# Mostramos las coleciones que hay en la base de datos actual:

> show collections
personas

# Podemos consultar los registros que contiene la colección personas con el comando db.NombreTabla.find():

> db.personas.find()
{ "_id" : ObjectId("5df4e71928fb000187f0d897"), "nombre" : "Pepito Perez", "localidad" : "Santiago de Compostela", "provincia" : "A
{ "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"), "nombre" : "Maria Yolanda Veiga", "localidad" : "Ourense", "provincia" : "Ourense" }
{ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"), "nombre" : "Jesus Fernandez", "localidad" : "Lugo", "provincia" : "Lugo" }

# Otra opcion
> db.personas.find().toArray()
[
```

```
        {
                "_id" : ObjectId("5df4e71928fb000187f0d897"),
                "nombre" : "Pepito Perez",
                "localidad" : "Santiago de Compostela",
                "provincia" : "A Coruña"
        },
        {
                "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"),
                "nombre" : "Maria Yolanda Veiga",
                "localidad" : "Ourense",
                "provincia" : "Ourense"
        },
        {
                "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"),
                "nombre" : "Jesus Fernandez",
                "localidad" : "Lugo",
                "provincia" : "Lugo"
        },
        {
                "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"),
                "nombre" : "Rafa",
                "localidad" : "Ourense",
                "provincia" : "Ourense"
        }
]

# O bien:

> db.personas.find().pretty()
{
        "_id" : ObjectId("5df4e71928fb000187f0d897"),
        "nombre" : "Pepito Perez",
        "localidad" : "Santiago de Compostela",
        "provincia" : "A Coruña"
}
{
        "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"),
        "nombre" : "Maria Yolanda Veiga",
        "localidad" : "Ourense",
        "provincia" : "Ourense"
}
{
        "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"),
        "nombre" : "Jesus Fernandez",
        "localidad" : "Lugo",
        "provincia" : "Lugo"
}
{
        "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"),
        "nombre" : "Rafa",
        "localidad" : "Ourense",
        "provincia" : "Ourense"
}
>

# Si queremos limitar el número de registros en la consulta, usamos la función limit():
> db.personas.find().limit(2).toArray()
{
        "_id" : ObjectId("5df4e71928fb000187f0d897"),
        "nombre" : "Pepito Perez",
        "localidad" : "Santiago de Compostela",
        "provincia" : "A Coruña"
}
{
        "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"),
        "nombre" : "Maria Yolanda Veiga",
        "localidad" : "Ourense",
        "provincia" : "Ourense"
}

# Si queremos mostrar por ejemplo los registros de la provincia de Ourense:
> db.personas.find({"provincia":"Ourense"})
{ "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"), "nombre" : "Maria Yolanda Veiga", "localidad" : "Ourense", "provincia" : "Ourense" }
```

```
{ "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"), "nombre" : "Rafa", "localidad" : "Ourense", "provincia" : "Ourense" }

# Ó bien:

> db.personas.find({"provincia":"Ourense"}).toArray()
[
        {
                "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"),
                "nombre" : "Maria Yolanda Veiga",
                "localidad" : "Ourense",
                "provincia" : "Ourense"
        },
        {
                "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"),
                "nombre" : "Rafa",
                "localidad" : "Ourense",
                "provincia" : "Ourense"
        }
]

# Se pueden utilizar operadores en la consulta:

# Más información sobre operadores en consultas: https://docs.mongodb.com/manual/reference/operator/query/

# Por ejemplo personas cuya provincia comience por la letra O incluída:

> db.personas.find({"provincia":{$gte:"Ourense"}}).toArray()
[
        {
                "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"),
                "nombre" : "Maria Yolanda Veiga",
                "localidad" : "Ourense",
                "provincia" : "Ourense"
        },
        {
                "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"),
                "nombre" : "Rafa",
                "localidad" : "Ourense",
                "provincia" : "Ourense"
        }
]

# Podemos hacer consultas con una proyección (indicando qué columnas queremos proyectar/mostrar en el resultado con 1 o 0 false para

find(consulta, proyección)

# Por ejemplo:
> db.personas.find({},{"nombre":true,"localidad":true}).toArray()
[
        {
                "_id" : ObjectId("5df4e71928fb000187f0d897"),
                "nombre" : "Pepito Perez",
                "localidad" : "Santiago de Compostela"
        },
        {
                "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"),
                "nombre" : "Maria Yolanda Veiga",
                "localidad" : "Ourense"
        },
        {
                "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"),
                "nombre" : "Jesus Fernandez",
                "localidad" : "Lugo"
        },
        {
                "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"),
                "nombre" : "Rafa",
                "localidad" : "Ourense"
        }
]

# El campo _id saldrá siempre por defecto a no ser que lo desactivemos en la proyección:
```

```
> db.personas.find({},{"nombre":true,"localidad":true,"_id":false}).toArray()
[
        {
                "nombre" : "Pepito Perez",
                "localidad" : "Santiago de Compostela"
        },
        {
                "nombre" : "Maria Yolanda Veiga",
                "localidad" : "Ourense"
        },
        {
                "nombre" : "Jesus Fernandez",
                "localidad" : "Lugo"
        },
        {
                "nombre" : "Rafa",
                "localidad" : "Ourense"
        }
]
```

6.11 Actualizando datos en MongoDB

En general las modificaciones se hacen mediante:

db.collection.update(query, update, options)

Este método puede modificar un documento existente o múltiples documentos en una colección. Puede además modificar campos específicos de un documento o documentos o reemplazar un documento completo, dependiendo de los parámetros que se pongan en update.

Parámetros disponibles: https://docs.mongodb.com/manual/reference/method/db.collection.update/#update-parameter

$set: mantiene todos los campos actuales y añade nuevos si fuera necesario. **multi**: Si se pone a true, actualiza múltiples documentos que cumplan el criterio de búsqueda.

```
# query selecciona los elementos que se van a actualizar.
# update indica los cambios que se van a hacer
# options es una parte opcional

# '''ATENCION''': '''Por defecto update() actualiza solamente 1 documento y actualiza ese campo pero elimina el resto de campos:'''
# Es decir si hacemos ésto:
db.personas.update({"provincia":"Ourense"},{"provincia":"Pontevedra"})

# Solamente actualizará el primer documento que cumpla esas características, y eliminará el resto de campos del documento, quedando
> db.personas.find()
{ "_id" : ObjectId("5df4e71928fb000187f0d897"), "nombre" : "Pepito Perez", "localidad" : "Santiago de Compostela", "provincia" : "A
{ "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"), "provincia" : "Pontevedra" }
{ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"), "nombre" : "Jesus Fernandez", "localidad" : "Lugo", "provincia" : "Lugo" }
{ "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"), "nombre" : "Marta García", "localidad" : "Ourense", "provincia" : "Ourense" }

# Supongamos que queremos actualizar todos los documentos cuya provincia es Ourense y ponerles Pontevedra:
> db.personas.find()
{ "_id" : ObjectId("5df4e71928fb000187f0d897"), "nombre" : "Pepito Perez", "localidad" : "Santiago de Compostela", "provincia" : "A
{ "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"), "provincia" : "Pontevedra" }
{ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"), "nombre" : "Jesus Fernandez", "localidad" : "Lugo", "provincia" : "Lugo" }
{ "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"), "nombre" : "Marta García", "localidad" : "Ourense", "provincia" : "Ourense" }

# Si queremos que mantenga el resto de campos del documento hay que añadir la opcion $set, en otro caso eliminará el resto de campos

>  db.personas.update({"provincia":"Ourense"},{$set:{"provincia":"Pontevedra"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.personas.find()
{ "_id" : ObjectId("5df4e71928fb000187f0d897"), "nombre" : "Pepito Perez", "localidad" : "Santiago de Compostela", "provincia" : "A
{ "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"), "provincia" : "Pontevedra" }
{ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"), "nombre" : "Jesus Fernandez", "localidad" : "Lugo", "provincia" : "Lugo" }
{ "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"), "nombre" : "Marta García", "localidad" : "Ourense", "provincia" : "Pontevedra" }
```

Si queremos que se actualicen todos los documentos hay que añadir un parámetro. Ese parámetro es multi:true

Véase el ejemplo:

```
> db.personas.update({"provincia":"Ourense"},{$set:{"provincia":"Pontevedra"}},{multi:true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })

> db.personas.find()
{ "_id" : ObjectId("5df4e71928fb000187f0d897"), "nombre" : "Pepito Perez", "localidad" : "Santiago de Compostela", "provincia" : "A
{ "_id" : ObjectId("5df4e79ea2aacff91fbd67bb"), "provincia" : "Pontevedra" }
{ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"), "nombre" : "Jesus Fernandez", "localidad" : "Lugo", "provincia" : "Lugo" }
{ "_id" : ObjectId("5df4ed46a2aacff91fbd67bd"), "nombre" : "Marta García", "localidad" : "Ourense", "provincia" : "Pontevedra" }
```

6.12 Borrando datos en MongoDB

Para borrar elementos de una colección se puede usar el método **remove()** que recibe como parámetro una query y elimina todos los elementos de la colección que coinciden con la consulta.

Por ejemplo:

```
# Si queremos borrar todos los documentos de la provincia de Ourense:
> db.personas.remove({"provincia":"Pontevedra"})
WriteResult({ "nRemoved" : 2 })

> db.personas.find()
{ "_id" : ObjectId("5df4e71928fb000187f0d897"), "nombre" : "Pepito Perez", "localidad" : "Santiago de Compostela", "provincia" : "A
{ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc"), "nombre" : "Jesus Fernandez", "localidad" : "Lugo", "provincia" : "Lugo" }

# Si queremos eliminar un elemento por el ID:
>db.personas.remove({ "_id" : ObjectId("5df4e7a0a2aacff91fbd67bc")})
WriteResult({ "nRemoved" : 1 })

# Si queremos eliminar todos los documentos de la colección:
> db.personas.remove({})
WriteResult({ "nRemoved" : 2 })

> db.personas.find()
```

7 Creación de un servidor web en Nodejs

Una de las grandes ventajas de Nodejs es su simplicidad. A diferencia de PHP o ASP en NodeJS no hay separación entre el servidor web y el propio código de la aplicación, ni tampoco tenemos que editar grandes ficheros de configuración para definir un comportamiento concreto. Con Node creamos el servidor, lo configuramos y entregamos el contenido y todo a nivel de código.

Vamos a ver cómo crear un servidor web con Node y enviar contenido a través de él, con seguridad y eficiencia.

Para poder enviar contenido web necesitamos crear una URI (Universal Resource Identifier) que identifique el recurso al que queremos acceder. En otras palabras crear las diferentes rutas de la aplciación que mostrarán las diferentes páginas o contenidos de la web.

Lo primero que haremos será crear nuestro fichero de servidor. Por ejemplo le llamamos server.js

El proceso consistiría en:

1. **Escribir todo el código del servidor web y de nuestra aplicación dentro del fichero server.js.**
2. **Guardar el fichero server.js**
3. **Arrancar la aplicación con el comando: node server.js**
4. **Y así una y otra vez, cada vez que hagamos modificaciones en el fichero.**

Para evitar el tener que ejecutar el comando: node server.js cada vez que modifiquemos la aplicación, podemos utilizar una aplicación adicional llamada **hotnode**, que se encargará de reiniciar nuestra aplicación cada vez que modifiquemos el fichero server.js.

Instalamos hotnode de forma global, para utilizarlo más adelante con nuestro server:

```
# Instalamos hotnode globalmente
npm -g install hotnode

# Para ejecutar el servidor:
```

```
hotnode server.js

# Hotnode auto-reiniciará nuestro servidor cada vez que grabemos las modificaciones en el fichero.
```

Para crear el servidor web necesitamos un módulo denominado http, que instalaremos dentro de la carpeta dónde tendremos nuestro servidor:

```
# Instalamos el módulo http:
npm install http

# Código que va dentro del fichero server.js:

var http = require('http');
http.createServer(function (request, response)
{
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end('Funciona mi servidor con Node.js!');
}).listen(8080);

# Si estamos utilizando el VPS de Google tendremos que abrir en el firewall el puerto 8080 para permitir dicho tráfico al servidor:
# Lo haremos en:

Google Cloud Platform -> Red de VPC -> Reglas de cortafuegos

# Agregaremos una regla:
# Nombre: nodejs
# Descripción:  Para acceder al servidor de Node en 8080
# Destinos: Todas las instancias de la red
# Intervalos de IPs de origen: 0.0.0.0/0
# Protocolos y puertos:
#  tcp: 8080
# Pulsamos el boton Crear.


# Para comprobar si funciona nuestro servidor web, nos tendremos que conectar a nuestro dominio pero al puerto 8080.
# Por ejemplo:

http://veiga.dynu.net:8080/

# Y se mostrará el mensaje:
Funciona mi servidor con Node.js!
```

8 Aplicación y API REST con MongoDB y NodeJS

Vamos a crear una aplicación con NodeJS que utilice MongoDB para almacenar información.

Para ello vamos a crear primeramente la base de datos, usuarios y tablas en nuestro servidor de **MongoDB**.

Si no queremos mantener nuestro propio servidor de **MongoDB** podemos utilizar algún **servidor gratuito que nos facilita hasta 500MB de almacenamiento** para nuestra base de datos.

Véase por ejemplo en: **https://mlab.com/**

8.1 Creación de base de datos y tablas en nuestro servidor de MongoDB

8.1.1 Creación de la base de datos tienda y usuario tienda en MongoDB

```
# Nos conectamos al shell de mongo
mongo

# Nos conectamos a la base de datos tienda:
> use tienda
switched to db tienda

# Comprobamos que estamos en la base de datos tienda:
> db
tienda

# Creamos un usuario tienda con contraseña abc123. y con permisos de propietario de la base de datos (dbOwner:
```

```
> db.createUser(
... {
... user: "tienda",
... pwd: "abc123.",
... roles: ["dbOwner"]
... }
... )
Successfully added user: { "user" : "tienda", "roles" : [ "dbOwner" ] }
```

8.1.2 Creación de las colecciones en MongoDB

```
# Primeramente vamos a comenzar creando una colección con unos cuantos productos.
# Comprobamos que estamos en la base de datos tienda:
> db
tienda

# A la hora de insertar en la coleccion productos, si ésta no existe se creará y almacenará el documento que estamos insertando.
# Podemos insertar los documentos de 1 en 1 con el comando:

> db.productos.insert({"nombre":"ColaCao Noir","precio": 2.35 })
WriteResult({ "nInserted" : 1 })

# O bien podemos insertar todos los documentos en una única instrucción:
> db.productos.insertMany( [
    { nombre: "Vileda Fregona", precio: 3.90 },
    { nombre: "Sanytol", precio: 2.40 },
    { nombre: "Tenn Limpiador", precio: 1.79 },
    { nombre: "Finish Powerball", precio: 15.60 },
    { nombre: "Estrella detergente", precio: 1.54 },
    { nombre: "Don Limpio", precio: 2.65 },
    { nombre: "KH-7", precio: 2.68 },
    { nombre: "Ariel 3 en 1", precio: 30.89 },
    ] );

{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5df6565b6be6d8a85f799e73"),
                ObjectId("5df6565b6be6d8a85f799e74"),
                ObjectId("5df6565b6be6d8a85f799e75"),
                ObjectId("5df6565b6be6d8a85f799e76"),
                ObjectId("5df6565b6be6d8a85f799e77"),
                ObjectId("5df6565b6be6d8a85f799e78"),
                ObjectId("5df6565b6be6d8a85f799e79"),
                ObjectId("5df6565b6be6d8a85f799e7a")
        ]
}

# Podemos ver todos los productos insertados:
> db.productos.find()
{ "_id" : ObjectId("5df6562b6be6d8a85f799e72"), "nombre" : "ColaCao Noir", "precio" : 2.35 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e73"), "nombre" : "Vileda Fregona", "precio" : 3.9 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e74"), "nombre" : "Sanytol", "precio" : 2.4 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e75"), "nombre" : "Tenn Limpiador", "precio" : 1.79 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e76"), "nombre" : "Finish Powerball", "precio" : 15.6 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e77"), "nombre" : "Estrella detergente", "precio" : 1.54 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e78"), "nombre" : "Don Limpio", "precio" : 2.65 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e79"), "nombre" : "KH-7", "precio" : 2.68 }
{ "_id" : ObjectId("5df6565b6be6d8a85f799e7a"), "nombre" : "Ariel 3 en 1", "precio" : 30.89 }
```

8.2 Carpeta inicial del proyecto e inicialización con npm init

Primeramente nos crearemos una carpeta e inicializaremos el proyecto:

```
# Crearemos la carpeta tiendaapp dónde queramos.
# Yo lo hago en una ruta que me permita subir ficheros con SublimeText al servidor:
mkdir tiendaapp
cd tiendaapp

# Inicializamos el proyecto con npm init.
# Al terminar el asistente, se creará un fichero package.json. En el fichero se almacena la información de los paquetes locales nece
```

```
npm init

This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (tiendaapp )
version: (1.0.0)
description: Demo aplicacion con MongoDB y NodeJS
entry point: (index.js)
test command:
git repository:
keywords:
author: Rafa Veiga
license: (ISC)
About to write to /var/www/veiga.dynu.net/public/nodejs/tiendaapp/package.json:

{
  "name": "tiendaapp",
  "version": "1.0.0",
  "description": "Demo aplicacion con MongoDB y NodeJS",
  "main": "index.js",
  "directories": {
    "doc": "docs"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Rafa Veiga",
  "license": "ISC"
}


Is this OK? (yes)


# Veamos el contenido del fichero package.json:
cat package.json

{
  "name": "tiendaapp",
  "version": "1.0.0",
  "description": "Demo aplicacion con MongoDB y NodeJS",
  "main": "index.js",
  "directories": {
    "doc": "docs"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Rafa Veiga",
  "license": "ISC"
}
```

8.3 Instalación de paquetes adicionales para la aplicación

Vamos a necesitar los paquetes adicionales de Node para el funcionamiento de nuestra aplicación:

1. ExpressJS: Este módulo de Node nos permitirá trabajar con un servidor web, rutas, etc..
2. Mongoose: El módulo que nos permitirá trabajar con bases de datos MongoDB
3. body-parser: Este módulo nos permite gestionar peticiones de tipo JSON.

```
# Podemos instalar las aplicaciones a través de la línea de comandos, de forma que automáticamente modifiquen también el fichero pac
npm install --save express mongoose body-parser
```

```
# Ésto instalará los 3 módulos en el direcotrio node_modules y modificará el fichero package.json tal y como se muestra:
cat package.json

{
  "name": "productosapp",
  "version": "1.0.0",
  "description": "Demo aplicacion con MongoDB y NodeJS",
  "main": "index.js",
  "directories": {
    "doc": "docs"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Rafa Veiga",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "mongoose": "^5.8.2"
  }
}
```

8.4 Inicializando el servidor

En el fichero **package.json** indicamos que la aplicación principal se llamaba index.js, así que vamos a crear el contenido de dicha aplicación.

Contenido del fichero index.js:

```
// index.js

// Cargamos el módulo express
const express = require('express');

// Cargamos el módulo body-parser
const bodyParser = require('body-parser');


// Inicializamos nuestra aplicación
const app = express();

// Configuramos el puerto que usaremos para nuestro servidor web de Nodejs
// Pondremos el 8080 que es el puerto que tenemos abierto en el firewall del servidor de Google.

let port = 8080;

app.listen(port, () => {
    console.log('Servidor web funcionando en el puerto ' + port);
});
```

Para arrancar la aplicación lo haremos desde la línea de comandos con: **node index.js**

```
# Arrancamos la aplicación:
node index.js

# En la línea de comandos se mostrará:
Servidor web funcionando en el puerto 8080

# Si nos conectamos con un navegador a la URL de nuestro servidor de Google (http://veiga.dynu.net:8080 ) y al puerto 8080, veremos
Cannot GET /

# Ya que nuestro servidor web no está enviando ningún tipo de información, pero sí que está escuchando en el puerto 8080, como podem
Servidor web funcionando en el puerto 8080

# Si el servidor web no estuviera funcionando, nuestro navegador web cliente, mostraría un error del estilo:

No se puede acceder a este sitio web
La página veiga.dynu.net ha rechazado la conexión.
ERR_CONNECTION_REFUSED
```

8.5 Rearranque automático del servidor de Node.js en modo de desarrollo

Cuando estamos desarrollando una aplicación, cada vez que se produzca una edición, si queremos probar los cambios, tendremos que parar la ejecución del script con **CTRL+C** y arrancar de nuevo la aplicación con **node servidor.js**.

Para evitar tener que hacer ésto constantemente, podemos utilizar una herramienta (**nodemon**) que se encargarían de **monitorizar si hay cambios en la aplicación y reiniciarla** si fuera necesario.:

- **nodemon**: **https://www.npmjs.com/package/nodemon**

Podemos **instalar nodemon de forma global** con el siguiente comando:

```
npm install -g nodemon

# Para monitorizar el fichero servidor.js haremos:
nodemon servidor.js

[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Servidor funcionando en el puerto: 8080
Conectado a servidor MongoDB.
```

8.6 Ejecución permanente de un script de Node.js en producción

- Cuando ya tenemos el servidor en **producción** funcionando correctamente, podemos emplear una herramienta que se llama **forever**, que se encarga de monitorizar la aplicación y si ésta se para por alguna razón, volver a arrancarla de forma automática.
- Atención **forever no monitoriza los cambios en la aplicación**, solamente se encarga de mantenerla en funcionamiento permanente.
- Más información sobre **forever** en: **https://www.npmjs.com/package/forever**

Podemos **instalar forever de forma global** con el siguiente comando:

```
npm install -g forever

# Para arrancar de forma continua el fichero index.js haremos:
forever start index.js

# Para ver la lista de procesos controlados por forever:
forever list

# Para parar un proceso:
forever stop uid_del_proceso

# Para reiniciar todos los procesos:
forever restartall

# Si queremos que nuestro script de Node, se arranque cuando se inicia la máquina de Linux en Debian, haremos por ejemplo un script
nano arrancarservidornode.sh

# Con el siguiente contenido, por ejemplo:
----------------------------------------------------------------

#!/bin/bash
clear
export PATH=/usr/bin:$PATH
RUTA="/var/www/veiga.dynu.net/nodejs/bottelegram/"
LOG="server.log"

cd $ruta
# Inicializamos el log al arrancar el equipo:
echo "" > $RUTA$LOG

forever start -o $RUTA$LOG --command node --sourceDir $RUTA index.js
```

```
-----------------------------------------------------------------

# Guardamos el fichero arrancarservidornode.sh
# Le aplicamos permisos:
chmod 755 arrancarservidornode.sh

# A continuación metemos el script bash en el arranque del sistema:
crontab -e

# Añadimos las 2 siguientes líneas:

# Arranque de servidor de node con forever
@reboot /ruta_del_script_/arrancarservidornode.sh

# Salimos guardando los cambios.

# Desactivamos NGINX en el arranque para evitar que ocupe el puerto 80
systemctl disable nginx

# Probamos a reiniciar el servidor:
reboot

# Y comprobaremos que el proceso de node está funcionando al reiniciar el sistema:
sudo su
forever list

# ATENCION: Si queremos volver a habilitar NGINX en el arranque pondremos:
systemctl enable nginx
```

8.7 Estructura de directorios de la aplicación

Trabajaremos con el patrón de diseño **MVC (Modelo Vista Controlador)**.

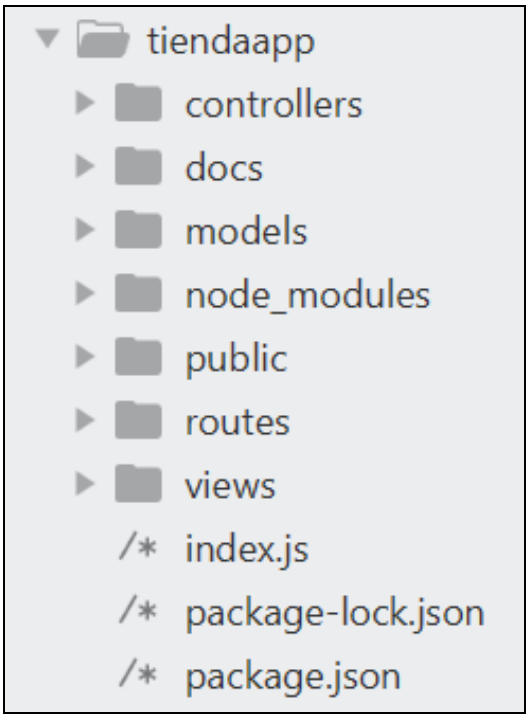
De esta forma separaremos las partes de nuestra aplicación y las agruparemos en base a su funcionalidad.

- **M** - viene de **Modelo**: aquí incluiremos todo el código que tenga que ver con la gestión y modelado de la base de datos (Tienda en nuestro caso).
- **V** - viene de **Vista**: aquí se incluirán las plantillas o vistas que nos permitirán mostrar información, formularios, listados, etc.. al cliente.
- **C** - viene de **Controlador**: aquí se incluye el código referente a la lógica de la aplicación. Esta lógica se encarga de gestionar las peticiones y enviar las respuestas. Además aquí vamos a incluir las Rutas. Las Rutas son la guía que le indican al cliente (navegador/aplicación móvil) a qué controlador dirigirse en base a la URL/ruta solicitada.

Dentro de la carpeta **tiendaapp**, crearemos los siguientes **subdirectorios**.

1. **controllers** Controladores de la aplicación
2. **models** Modelos de la aplicación
3. **routes** Rutas de la aplicación
4. **views** Vistas de la aplicación
5. **public** Recursos públicos de la aplicación: librerías JavaScript, ficheros JavaScript, CSS, etc..

Ahora ya tenemos el servidor preparado para gestionar las peticiones y algunos subdirectorios para colocar ahí nuestro código.



8.7.1 La aplicación principal index.js

Ésta será el contenido de la aplicación principal **index.js**

```
//index.js

const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const api = require('./routes/api.route'); // Importamos las rutas de la futura API que están en ./routes/api.route.js

// https://expressjs.com/en/4x/api.html#app.use
// Le indicamos a la aplicación que al poner /api/v1/productos, llame al middleware api, que en este caso es  un middleware que se e
app.use('/api/v1/productos', api);

// Configuramos el puerto de escucha de nuestro servidor web.
// Asegurarse de abrir puerto en el servidor de Google, Amazon, etc...
let port = 8080;
app.listen(port, () => {
    console.log('Servidor funcionando en el puerto: ' + port);
});
```

8.7.2 El Modelo

Vamos a comenzar definiendo nuestro modelo.

Para ello crearemos un nuevo fichero dentro del subdirectorio models y lo nombraremos como **productos.model.js**

```
// Cargamos el módulo de MongoDB
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

// Definimos el schema para una colección llamada Producto
let ProductoSchema = new Schema({
    nombre: {type: String, required: true, max: 100},
    precio: {type: Number, required: true},
});

// Exportamos el modelo
module.exports = mongoose.model('Producto', ProductoSchema);
```

8.7.3 Las Rutas

Vamos a pensar cómo serían las URLs en nuestra aplicación, pensando en una futura API REST que desarrollaremos más adelante.

En una **API REST** se suelen definir las rutas incluyendo el **versionado**. En este caso nuestra futura API REST será una ruta del estilo: **/api/v1/productos**

Para ello diseñaremos las rutas,dentro del subdirectorio **routes** creando un fichero **api.route.js**.

Algunos desarrolladores prefieren poner todas las rutas en un único fichero llamado routes.js, pero ésto no es muy recomendable si tenemos una aplicación que gestione muchas rutas o vaya a crecer.

Contenido del fichero **api.route.js**:

```
// api.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const api_controller = require('../controllers/api.controller');

// Ejemplo de prueba de una ruta por GET a la URL /api/v1/productos/test
// Aquí le estamos indicando que cuando pongamos la ruta /api/productos/test que llame al controlador api_controller y a la función
/* La comentamos para no interferir con otras rutas de tipo GET */
router.get('/test', api_controller.api_test);

module.exports = router;
```

8.7.4 Los Controladores

El siguiente paso será implementar los controladores, a los que hicimos referencia en la sección de Rutas.

Dentro del subdirectorio controllers, crearemos un nuevo fichero llamado **api.controller.js** dónde programaremos los controladores.

```
// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /api/productos/test
exports.api_test = function (req, res) {
    res.send('Saludos desde el controlador de pruebas /api/v1/productos/test !');
};
```

8.7.5 Prueba inicial de la ruta test

Ahora podremos probar la aplicación entrando en nuestra URL:puerto/api/v1/productos/test

Para ello arrancamos el servidor:

```
node index.js
```

Y nos conectamos con un navegador web a: **http://veiga.dynu.net:8080/api/v1/productos/test**

Y nos tendrá que mostrar algo como:

Saludos desde el controlador de pruebas /api/v1/productos/test !

8.8 Conectando la aplicación a la base de datos tienda en MongoDB

Ahora tenemos que informar a nuestra aplicación que se tiene que comunicar con la base de datos que hemos creado en MongoDB.

Para ello hacemos uso del módulo Mongoose que instalamos al principio.

Lo que tendremos que hacer será indicar en nuestra aplicación **index.js** y pegar el siguiente código adaptado a nuestra conexión:

```
//index.js

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

const api = require('./routes/api.route'); // Importamos las rutas de la futura API

// https://expressjs.com/en/4x/api.html#app.use
// Le indicamos a la aplicación que al poner /api/v1/productos, dichas rutas las va a gestionar api.route.js
app.use('/api/v1/productos', api);

///////////////// MOONGOOSE /////////////////////

// Definimos las opciones de conexión a MongoDB:
const opcionesMongo = {
    keepAlive: 1,
    useUnifiedTopology: true,
    useNewUrlParser: true
  };

var mongoose = require('mongoose');

// Ajustamos la conexión a nuestro servidor MongoDB
// mongoose.connect('mongodb://username:password@host:port/database');
// Cuando nos conectamos si hay un error en la autenticación o el servidor está caído, a los 10 segundos aparecerá el error en la co
var mongoDB = 'mongodb://tienda:abc123.@127.0.0.1:27017/tienda';

// Realizamos la conexión inicial
mongoose.connect(mongoDB,opcionesMongo).catch(error => { console.log('No me puedo conectar al servidor MongoDB: '+error)} );

// Programamos el evento de error para que se dispare si se produce algún error después de la conexión inicial con la base de datos
mongoose.connection.on('error', error => { console.log('Se ha producido un error en conexión a MongoDB: '+error)} );
mongoose.connection.on('connected', () => { console.log('Conectado a servidor MongoDB.')} );

///////////////// FIN SECCION MOONGOOSE /////////////////////

// Configuramos el puerto de escucha de nuestro servidor web.
// Asegurarse de abrir puerto en el servidor de Google, Amazon, etc...
let port = 8080;
app.listen(port, () => {
    console.log('Servidor funcionando en el puerto: ' + port);
});
```

8.9 Configurando el módulo bodyParser

Nuestra aplicación usa el middleware **bodyParser** que instalamos al principio y que se encargará de gestionar todos los datos que llegan a la aplicación.

ATENCIÓN: Tenemos que tener la precaución, de c**onfigurar dicho módulo antes de la importación de las rutas**.

Para ello tendremos que integrarlo dentro de la aplicación principal añadiendo lo siguiente a continuación de express():

```
// Parte de bodyParser.
app.use(bodyParser.json()); // Para parsear application/json
app.use(bodyParser.urlencoded({extended: false})); // Para parsear application/x-www-form-urlencoded
```

Fichero **index.js** completo:

```
//index.js
```

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

// Parte de bodyParser, que incluiremos antes de la importación de las rutas.
app.use(bodyParser.json()); // Para parsear application/json
app.use(bodyParser.urlencoded({extended: false})); // Para parsear application/x-www-form-urlencoded

const api = require('./routes/api.route'); // Importamos las rutas de la futura API

// https://expressjs.com/en/4x/api.html#app.use
// Le indicamos a la aplicación que al poner /api/v1/productos, dichas rutas las va a gestionar api.route.js
app.use('/api/v1/productos', api);

///////////////// MOONGOOSE ////////////////////

// Definimos las opciones de conexión a MongoDB:
const opcionesMongo = {
    keepAlive: 1,
    useUnifiedTopology: true,
    useNewUrlParser: true
  };

var mongoose = require('mongoose');

// Ajustamos la conexión a nuestro servidor MongoDB
// mongoose.connect('mongodb://username:password@host:port/database');
// Cuando nos conectamos si hay un error en la autenticación o el servidor está caído, a los 10 segundos aparecerá el error en la co
var mongoDB = 'mongodb://tienda:abc123.@127.0.0.1:27017/tienda';

// Realizamos la conexión inicial
mongoose.connect(mongoDB,opcionesMongo).catch(error => { console.log('No me puedo conectar al servidor MongoDB: '+error)} );

// Programamos el evento de error para que se dispare si se produce algún error después de la conexión inicial con la base de datos
mongoose.connection.on('error', error => { console.log('Se ha producido un error en conexión a MongoDB: '+error)} );
mongoose.connection.on('connected', () => { console.log('Conectado a servidor MongoDB.')} );

///////////////// FIN SECCION MOONGOOSE ////////////////////

// Configuramos el puerto de escucha de nuestro servidor web.
// Asegurarse de abrir puerto en el servidor de Google, Amazon, etc...
let port = 8080;
app.listen(port, () => {
    console.log('Servidor funcionando en el puerto: ' + port);
});
```

8.10 Implementación de las operaciones de la API REST

- A la hora de crear la API REST de nuestra aplicación, necesitamos crear las URI o identificadores únicos (endpoints) de nuestra API.
- Estas URI serán las encargadas de recibir las peticiones y de devolver los resultados u operaciones solicitadas.
- Vamos a realizar la implementación de las **operaciones CRUD a través de la API REST**.
- Más información y conceptos sobre API REST en: **https://asiermarques.com/2013/conceptos-sobre-apis-rest/**

En una **API REST** se usan los siguientes verbos HTTP:

- **GET**: Para consultar y leer recursos
- **POST**: Para crear recursos
- **PUT**: Para editar recursos
- **DELETE**: Para eliminar recursos.
- **PATCH**: Para editar partes concretas de un recurso.

8.10.1 CREATE

La primera tarea CRUD en nuestra API REST será la de generar una ruta que nos permita **crear un nuevo producto en la tienda imaginaria**.

Para ello comenzamos definiendo primero la **ruta** en la API que se encargará de realizar dicho proceso.

La programaremos en el fichero que hemos diseñado para las rutas de la API: **routes/api.route.js**

```
// api.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const api_controller = require('../controllers/api.controller');

// Ejemplo de prueba de una ruta por GET a la URL /api/v1/productos/test
// Aquí le estamos indicando que cuando pongamos la ruta /api/productos/test que llame al controlador api_controller y a la función
/* La comentamos para no interferir con otras rutas de tipo GET */
// router.get('/test', api_controller.api_test);

// Operación de CREATE. Se realizará usando el método HTTP POST a la ruta /api/productos/create que programamos dentro de la API.
router.post('/create', api_controller.api_productos_create);

module.exports = router;
```

A continuación una vez creada la ruta por **POST** a **/api/create**, definimos la **lógica** en el controlador **api.controller.js**.

Acordarse de que tenemos que reiniciar el servidor con cada modificación que hagamos.

```
// api.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /api/v1/productos/test
exports.api_test = function (req, res) {
    res.send('Saludos desde el controlador de pruebas /api/v1/productos/test !');
};


exports.api_productos_create = function (req, res) {
    let producto = new Producto(
    {
        nombre: req.body.nombre,
        precio: req.body.precio
    }
    );

    producto.save(function (err) {
        if (err)
        {
            // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
            return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!"
        }

        // Enviamos al cliente la siguiente respuesta con el código HTTP 200.
        res.type('json').status(200).send({ status: "ok", data: "Producto creado satisfactoriamente!" });
    })
};
```

Probamos a ver si funciona la ruta API de create usando la aplicación cliente "Advanced REST Client" de Chrome:

Request

Method
POST

Request URL
http://veiga.dynu.net:8080/api/v1/productos/crea

Parameters

Headers | Body

Body content type
application/x-www-for...

Editor view
Form data (www-url-form-encodec

ENCODE PAYLOAD DECODE PAYLOAD

nombre	Detergen
precio	5.80

ADD FORM PARAMETER

200 OK 315.19 ms


```
{
    "status": "ok",
    "data": "Producto creado satisfactoriamente!"
}
```

Comprobamos en la base de datos si aparece el **nuevo documento que hemos insertado al final**:

```
mongo

> use tienda
switched to db tienda

> show collections
productos

> db.productos.find()
{ "_id" : ObjectId("5df67753afbcb56eeefe1039"), "nombre" : "ColaCao Noir", "precio" : 2.35 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103a"), "nombre" : "Vileda Fregona", "precio" : 3.9 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103b"), "nombre" : "Sanytol", "precio" : 2.4 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103c"), "nombre" : "Tenn Limpiador", "precio" : 1.79 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103d"), "nombre" : "Finish Powerball", "precio" : 15.6 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103f"), "nombre" : "Don Limpio", "precio" : 2.65 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe1040"), "nombre" : "KH-7", "precio" : 2.68 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe1041"), "nombre" : "Ariel 3 en 1", "precio" : 30.89 }
{ "_id" : ObjectId("5df6776c3891d34bb6a4d001"), "nombre" : "Detergente Skip", "precio" : 7.85, "__v" : 0 }
{ "_id" : ObjectId("5dff7c4fcd2ca8703312aaf8"), "nombre" : "Detergente Colón", "precio" : 5.8, "__v" : 0 }
```

8.10.2 READ

- Vamos ahora a realizar la tarea de **lectura de productos** a través de la API REST.
- En este caso tendremos que crear una ruta de tipo **GET** a la raíz /api/productos en el fichero de rutas **api.route.js**.
- Lo programaremos de tal forma, que si se le pasa algún ID devolverá los datos de ese producto, en otro caso devolverá todos los productos de la colección.

Para ello comenzamos definiendo primero la ruta en el fichero **routes/api.route.js**

```
// api.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const api_controller = require('../controllers/api.controller');

// Ejemplo de prueba de una ruta por GET a la URL /api/v1/productos/test
// Aquí le estamos indicando que cuando pongamos la ruta /api/productos/test que llame al controlador api_controller y a la función
/* La comentamos para no interferir con otras rutas de tipo GET */
// router.get('/test', api_controller.api_test);

// Operación de CREATE. Se realizará usando el método HTTP POST a la ruta /api/productos/create que programamos dentro de la API.
router.post('/create', api_controller.api_productos_create);

// Operación de READ. Se realizará usando el método HTTP GET a la ruta /api/v1/productos que programamos dentro de la API
// El parámetro id es opcional. LLamará a la función api_productos_read del controlador.
router.get('/:id?', api_controller.api_productos_read);

module.exports = router;
```

Ahora programamos la **funcionalidad** de dicha ruta en el **controlador** de productos en **api.controller.js**:

```
// api.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /api/v1/productos/test
exports.api_test = function (req, res) {
    res.send('Saludos desde el controlador de pruebas /api/v1/productos/test !');
};
```

```
exports.api_productos_create = function (req, res) {
    let producto = new Producto(
    {
        nombre: req.body.nombre,
        precio: req.body.precio
    }
    );

    producto.save(function (err) {
        if (err)
        {
            // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
            return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!"
        }

        // Enviamos al cliente la siguiente respuesta con el código HTTP 200.
        res.type('json').status(200).send({ status: "ok", data: "Producto creado satisfactoriamente!" });
    })
};

exports.api_productos_read = function (req, res) {
    // Si le pasamos un ID de producto devuelve los datos de ese producto
    // En otro caso devolverá todos los productos
    if (req.params.id)
    {
        Producto.findById(req.params.id, function (err, unProducto) {
            if (err)
            {
                // Devolvemos el código HTTP 404, de producto no encontrado por su id.
                res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
            }
            else
            {
                // También podemos devolver así la información:
                res.status(200).json({ status: "ok", data: unProducto });
            }
        })
    }
    else
    {
        // Ayuda: https://mongoosejs.com/docs/api/model.html
        Producto.find({}, function (err, todosProductos) {
            if (err)
          // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
          return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!" })

            // También podemos devolver así la información:
            res.status(200).json({ status: "ok", data: todosProductos });
        })
    }
};
```

Capturas de la prueba de funcionamiento pasando un ID de producto y sin pasar ningún ID:

Request

Method
GET

Request URL
http://veiga.dynu.net:8080/api/v1/productos/5df6

Parameters

Headers

Toggle source mode Insert headers set

Header name	Header value
Content-Type	application/x-www-form-urlencoded

ADD HEADER

200 OK 315.63 ms


```
{
    "status": "ok",
  - "data": {
        "_id": "5df6776c3891d34bb6a4d001",
        "nombre": "Detergente Skip",
        "precio": 7.85,
        "__v": 0
    }
```

Request

Method
GET

Request URL
http://veiga.dynu.net:8080/api/v1/productos

Parameters

Headers

Toggle source mode Insert headers set

Header name	Header value
Content-Type	application/x-www-form-urlencoded

ADD HEADER

200 OK 122.68 ms


```
{
    "status": "ok",
  - "data": [Array[10]
      - 0:  {
            "_id": "5df67753afbcb56eeefe1039",
            "nombre": "ColaCao Noir",
            "precio": 2.35
        },
```

8.10.3 UPDATE

Vamos ahora a realizar la tarea de **actualización de productos**.

- En este caso tendremos que crear una ruta de tipo **PUT** en el fichero de rutas **productos.route.js**, a la ruta /**api**/**v1**/**productos** pasando un ID del producto a actualizar.
- Lo programaremos de tal forma, que si se le pasa algún ID devolverá los datos de ese producto, en otro caso devolverá todos los productos de la colección.

Para ello comenzamos definiendo primero la ruta en el fichero **routes**/**api.route.js**

```
// api.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const api_controller = require('../controllers/api.controller');

// Ejemplo de prueba de una ruta por GET a la URL /api/v1/productos/test
// Aquí le estamos indicando que cuando pongamos la ruta /api/productos/test que llame al controlador api_controller y a la función
/* La comentamos para no interferir con otras rutas de tipo GET */
// router.get('/test', api_controller.api_test);

// Operación de CREATE. Se realizará usando el método HTTP POST a la ruta /api/productos/create que programamos dentro de la API.
router.post('/create', api_controller.api_productos_create);

// Operación de READ. Se realizará usando el método HTTP GET a la ruta /api/v1/productos que programamos dentro de la API
// El parámetro id es opcional. LLamará a la función api_productos_read del controlador.
router.get('/:id?', api_controller.api_productos_read);

// Operación de PUT (se actualizarán todos los datos). Si quisiéramos realizar una actualización parcial usaríamos el verbo PATCH.
// Se realizará a la ruta / (en este caso es /api/v1/productos) pasando el ID del producto a actualizar.
router.put('/:id', api_controller.api_productos_update);

module.exports = router;
```

Ahora programamos la **funcionalidad** de dicha ruta en el **controlador** de productos en **api.controller.js**:

```
// api.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /api/v1/productos/test
exports.api_test = function (req, res) {
    res.send('Saludos desde el controlador de pruebas /api/v1/productos/test !');
};


exports.api_productos_create = function (req, res) {
    let producto = new Producto(
    {
        nombre: req.body.nombre,
        precio: req.body.precio
    }
    );

    producto.save(function (err) {
        if (err)
        {
            // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
            return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!"
        }

        // Enviamos al cliente la siguiente respuesta con el código HTTP 200.
```

```
        res.type('json').status(200).send({ status: "ok", data: "Producto creado satisfactoriamente!" });
    })
};

exports.api_productos_read = function (req, res) {
    // Si le pasamos un ID de producto devuelve los datos de ese producto
    // En otro caso devolverá todos los productos
    if (req.params.id)
    {
        Producto.findById(req.params.id, function (err, unProducto) {
            if (err)
            {
                // Devolvemos el código HTTP 404, de producto no encontrado por su id.
                res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
            }
            else
            {
                // También podemos devolver así la información:
                res.status(200).json({ status: "ok", data: unProducto });
            }
        })
    }
    else
    {
        // Ayuda: https://mongoosejs.com/docs/api/model.html
        Producto.find({}, function (err, todosProductos) {
            if (err)
          // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
          return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!" })

            // También podemos devolver así la información:
            res.status(200).json({ status: "ok", data: todosProductos });
        })
    }
};


exports.api_productos_update = function (req, res) {
    Producto.findByIdAndUpdate(req.params.id, { $set: req.body }, function (err, productoActualizado) {
        if (err)
        {
            //res.send(err);
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
        }
        else
        {
            // Devolvemos el código HTTP 200.
            res.status(200).json({ status: "ok", data: productoActualizado });
        }
    });
};
```

Prueba de funcionamiento, pasando un código que no existe y pasando un código correcto.

Si no encuentra el documento devuelve un código **404** con el mensaje de que no se ha encontrado el producto.

Si actualiza correctamente devolverá el código **200**, con los antiguos datos del documento que actualizó.

Actualización incorrecta:

Request

Method

PUT

Request URL

http://veiga.dynu.net:8080/api/v1/productos/123

Parameters

Headers | Body

Toggle source mode

Insert headers set

Header name	Header value
Content-Type	application/x-www-form-urlencoded

ADD HEADER

404 Not Found 304.16 ms


```
{
    "status": "error",
    "data": "No se ha encontrado el producto con id: 123468"
}
```


Actualización correcta:

Request

Method
PUT

Request URL
http://veiga.dynu.net:8080/api/v1/productos/5df6

Parameters

Headers | Body

Body content type
application/x-www-for...

Editor view
Form data (www-url-form-encodec

ENCODE PAYLOAD DECODE PAYLOAD

nombre	Tambor
precio	6.90

ADD FORM PARAMETER

200 OK 286.16 ms


```
{
    "status": "ok",
  - "data": {
        "_id": "5df6776c3891d34bb6a4d001",
        "nombre": "Detergente Skip",
```

8.10.4 DELETE

Vamos ahora a realizar la tarea de **borrar un producto**.

En este caso tendremos que crear una ruta de tipo **DELETE** a la raíz / pasando un id del producto a borrar en el fichero de rutas **api.route.js**.

Lo programaremos de tal forma, que si se le pasa algún ID devolverá los datos de ese producto, en otro caso devolverá todos los productos de la colección.

Para ello comenzamos definiendo primero la ruta en el fichero **routes/api.route.js**

```
// api.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const api_controller = require('../controllers/api.controller');

// Ejemplo de prueba de una ruta por GET a la URL /api/v1/productos/test
// Aquí le estamos indicando que cuando pongamos la ruta /api/productos/test que llame al controlador api_controller y a la función
/* La comentamos para no interferir con otras rutas de tipo GET */
// router.get('/test', api_controller.api_test);

// Operación de CREATE. Se realizará usando el método HTTP POST a la ruta /api/productos/create que programamos dentro de la API.
router.post('/create', api_controller.api_productos_create);

// Operación de READ. Se realizará usando el método HTTP GET a la ruta /api/v1/productos que programamos dentro de la API
// El parámetro id es opcional. LLamará a la función api_productos_read del controlador.
router.get('/:id?', api_controller.api_productos_read);

// Operación de PUT (se actualizarán todos los datos). Si quisiéramos realizar una actualización parcial usaríamos el verbo PATCH.
// Se realizará a la ruta / (en este caso es /api/v1/productos) pasando el ID del producto a actualizar.
router.put('/:id', api_controller.api_productos_update);

// Operación de DELETE para borrar un documento de una colección.
// Se realizará a la ruta / (en este caso es /api/v1/productos) pasando el ID del producto a borrar.
router.delete('/:id', api_controller.api_productos_delete);

module.exports = router;
```

Ahora programamos la **funcionalidad** de dicha ruta en el **controlador** de productos en **api.controller.js**:

```
// api.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /api/v1/productos/test
exports.api_test = function (req, res) {
    res.send('Saludos desde el controlador de pruebas /api/v1/productos/test !');
};


exports.api_productos_create = function (req, res) {
    let producto = new Producto(
    {
        nombre: req.body.nombre,
        precio: req.body.precio
    }
    );

    producto.save(function (err) {
        if (err)
        {
            // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
```

```
            return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!"
        }

        // Enviamos al cliente la siguiente respuesta con el código HTTP 200.
        res.type('json').status(200).send({ status: "ok", data: "Producto creado satisfactoriamente!" });
    })
};


exports.api_productos_read = function (req, res) {
    // Si le pasamos un ID de producto devuelve los datos de ese producto
    // En otro caso devolverá todos los productos
    if (req.params.id)
    {
        Producto.findById(req.params.id, function (err, unProducto) {
            if (err)
            {
                // Devolvemos el código HTTP 404, de producto no encontrado por su id.
                res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
            }
            else
            {
                // También podemos devolver así la información:
                res.status(200).json({ status: "ok", data: unProducto });
            }
        })
    }
    else
    {
        // Ayuda: https://mongoosejs.com/docs/api/model.html
        Producto.find({}, function (err, todosProductos) {
            if (err)
          // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
          return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!" })

            // También podemos devolver así la información:
            res.status(200).json({ status: "ok", data: todosProductos });
        })
    }
};


exports.api_productos_update = function (req, res) {
    Producto.findByIdAndUpdate(req.params.id, { $set: req.body }, function (err, productoActualizado) {
        if (err)
        {
            //res.send(err);
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
        }
        else
        {
            // Devolvemos el código HTTP 200.
            res.status(200).json({ status: "ok", data: productoActualizado });
        }
    });
};


exports.api_productos_delete = function (req, res) {
    Producto.findByIdAndRemove(req.params.id, function(err, data) {
        if (err || !data) {
            //res.send(err);
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
        }
        else
        {
            // Devolvemos el código HTTP 204 cuando lo ha borrado (No Content). No devuelve contenido.
            res.status(204).json({ status: "ok", data: "Se ha eliminado correctamente el producto con id: "+req.params.id});


        }
```

```
        });
    };
```

Prueba de funcionamiento, pasando un código que no existe y pasando un código correcto.

- **Si no encuentra el documento** devuelve un **código 404** con el mensaje de que no se ha encontrado el producto.
- **Borrado incorrecto por que no se encuentra el ID a borrar:**

Request

Method
DELETE

Request URL
http://veiga.dynu.net:8080/api/v1/productos/1234

Parameters

Headers | Body

Body content type
application/x-www-for...

Editor view
Form data (www-url-form-encoded

ENCODE PAYLOAD DECODE PAYLOAD

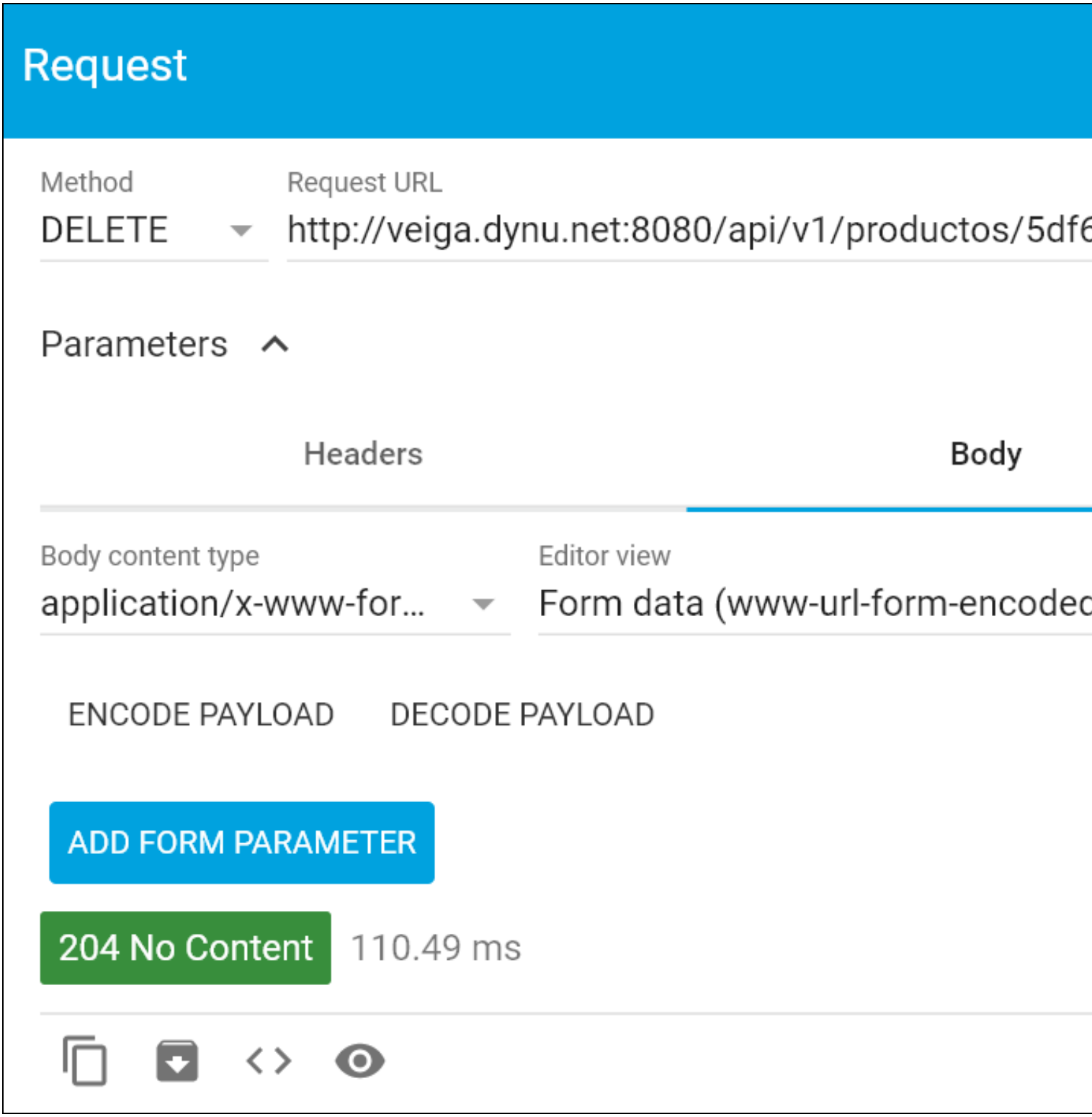
nombre	Tambor D
precio	6.90

ADD FORM PARAMETER

404 Not Found 128.89 ms


```
{
    "status": "error",
    "data": "No se ha encontrado el producto con id: 12348787"
}
```

- Ahora pasamos un código 5df6775cafbcb56eeefe103b del producto Sanytol, para comprobar que lo borra.
- **Borrado correcto. Notar que devuelve el** status HTTP 204 de No Content**, por lo que no se ve la respuesta que hemos programado.**
- Si quisiéramos ver información de respuesta, tendríamos que devolver código HTTP 200



Podemos comprobar en la base de datos que el producto con ID 5df6775cafbcb56eeefe103b ha desaparecido:

```
mongo

> use tienda
switched to db tienda

> db.productos.find()
{ "_id" : ObjectId("5df67753afbcb56eeefe1039"), "nombre" : "ColaCao Noir", "precio" : 2.35 }
```

```
{ "_id" : ObjectId("5df6775cafbcb56eeefe103a"), "nombre" : "Vileda Fregona", "precio" : 3.9 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103c"), "nombre" : "Tenn Limpiador", "precio" : 1.79 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103d"), "nombre" : "Finish Powerball", "precio" : 15.6 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe103f"), "nombre" : "Don Limpio", "precio" : 2.65 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe1040"), "nombre" : "KH-7", "precio" : 2.68 }
{ "_id" : ObjectId("5df6775cafbcb56eeefe1041"), "nombre" : "Ariel 3 en 1", "precio" : 30.89 }
{ "_id" : ObjectId("5df6776c3891d34bb6a4d001"), "nombre" : "Tambor Detergente Skip", "precio" : 6.9, "__v" : 0 }
{ "_id" : ObjectId("5dff7c4fcd2ca8703312aaf8"), "nombre" : "Detergente Colón", "precio" : 5.8, "__v" : 0 }
```

8.11 Desarrollo de frontend de nuesta aplicación con Node.js, MongoDB, Bootstrap y jQuery

- Vamos ahora a desarrollar un **frontend** que nos permitirá introducir datos en la base de datos, modificar y borrar desde un navegador web.
- Para ello tendremos que trabajar con un motor de plantillas que nos permitirá generar las vistas que se mostrarán al cliente.
- Podemos ver una lista completa de **Plantillas disponibles para ExpressJS** (módulo de servidor web de Node.js) en: **https://expressjs.com/en/resources/template-engines.html**
- Trabajaremos con un **nuevo archivo de rutas** para diferenciar las rutas de la API REST de las rutas de la aplicación principal.

8.11.1 Instalación de motor de plantillas Pug para Node.js

Vamos a trabajar con el motor de plantillas **Pug** : **https://pugjs.org/api/getting-started.html**

Para entender lo que hace Pug, necesitamos recordar que el navegador web lee HTML y CSS y muestra las imágenes y texto al cliente en el formato correspondiente que se indica con HTML y CSS.

Pug se sitúa en el medio. Es un gestor de plantillas para Node.js.

Un gestor de plantilla nos permite inyectar datos y generar el HTML correspondiente.

Brevemente: En tiempo de ejecución, Pug (y otros motores de plantillas) reemplazan las variables en nuestro fichero con los valores correspondientes, y envían el HTML resultante al cliente.

```
# Para instalar pug.
npm install pug --save

# Una vez instalado podremos comprobar en el fichero package.json que ya aparece referenciado:
cat package.json

{
  "name": "productosapp",
  "version": "1.0.0",
  "description": "Demo aplicacion con MongoDB y NodeJS",
  "main": "index.js",
  "directories": {
    "doc": "docs"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Rafa Veiga",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "mongoose": "^5.8.2",
    "pug": "^2.0.4"
  }
}
```

PDF con breve tutorial sobre motor de Plantillas Pug

8.11.2 Configuración de Pug en la aplicación

Una vez instalado el módulo Pug en nuestra aplicación, vamos a configurarla para indicarle que queremos usar dicho motor de plantillas y además le indicaremos el directorio dónde se encontrarán las plantillas que vayamos a utilizar.

```
// Ajustamos en nuestro index.js la configuración del motor de plantillas:
```

```
// añadiendo el siguiente código al principio después de const app = express();

----------------------------------
// Cargamos motor de plantillas
const pug = require("pug");

// Con Express no es necesario cargar el motor de plantillas,
// pero sí es necesario indicarle que lo estamos usando con:
app.set("view engine", "pug");

// Configuramos el directorio de las plantillas
app.set('views', __dirname + '/views');
---------------------------------------
```

El fichero index.js nos quedará tal que así:

```
//index.js

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

// Parte de bodyParser, que incluiremos antes de la importación de las rutas.
app.use(bodyParser.json()); // Para parsear application/json
app.use(bodyParser.urlencoded({extended: false})); // Para parsear application/x-www-form-urlencoded

const api = require('./routes/api.route'); // Importamos las rutas de la futura API

// https://expressjs.com/en/4x/api.html#app.use
// Le indicamos a la aplicación que al poner /api/v1/productos, dichas rutas las va a gestionar api.route.js
app.use('/api/v1/productos', api);

// Cargamos motor de plantillas
const pug = require("pug");

// Con Express no es necesario cargar el motor de plantillas,
// pero sí es necesario indicarle que lo estamos usando con:
app.set("view engine", "pug");

// Configuramos el directorio de las plantillas
app.set('views', __dirname + '/views');


//////////////// MOONGOOSE ////////////////////

// Definimos las opciones de conexión a MongoDB:
const opcionesMongo = {
    keepAlive: 1,
    useUnifiedTopology: true,
    useNewUrlParser: true,
    useFindAndModify: false
  };

var mongoose = require('mongoose');

// Ajustamos la conexión a nuestro servidor MongoDB
// mongoose.connect('mongodb://username:password@host:port/database');
// Cuando nos conectamos si hay un error en la autenticación o el servidor está caído, a los 10 segundos aparecerá el error en la co
var mongoDB = 'mongodb://tienda:abc123.@127.0.0.1:27017/tienda';

// Realizamos la conexión inicial
mongoose.connect(mongoDB,opcionesMongo).catch(error => { console.log('No me puedo conectar al servidor MongoDB: '+error)} );

// Programamos el evento de error para que se dispare si se produce algún error después de la conexión inicial con la base de datos
mongoose.connection.on('error', error => { console.log('Se ha producido un error en conexión a MongoDB: '+error)} );
mongoose.connection.on('connected', () => { console.log('Conectado a servidor MongoDB.')} );

//////////////// FIN SECCION MOONGOOSE ////////////////////

// Configuramos el puerto de escucha de nuestro servidor web.
// Asegurarse de abrir puerto en el servidor de Google, Amazon, etc...
let port = 8080;
app.listen(port, () => {
```

```
    console.log('Servidor funcionando en el puerto: ' + port);
});
```

8.11.3 Configuración de nueva ruta de la aplicación

- Para diferenciar la ruta de la **API REST** (**/api/v1/productos**) de la aplicación principal, definiremos una nueva ruta que gestionaremos con el fichero **aplicacion.route.js**.
- Añadiremos debajo de la línea: **app.use('/api/v1/productos', api)** el siguiente código:

```
// index.js
....
const aplicacion = require('./routes/aplicacion.route'); // Rutas dónde definimos las URL de la aplicación principal
// Ruta de la aplicación principal que apuntará a la RAIZ de la URL de nuestro dominio: /
app.use('/',aplicacion)
....
```

Quedando el fichero index.js de la siguiente forma:

```
//index.js

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

// Parte de bodyParser, que incluiremos antes de la importación de las rutas.
app.use(bodyParser.json()); // Para parsear application/json
app.use(bodyParser.urlencoded({extended: false})); // Para parsear application/x-www-form-urlencoded

const api = require('./routes/api.route'); // Importamos las rutas de la futura API
// Le indicamos a la aplicación que al poner /api/v1/productos, dichas rutas las va a gestionar api.route.js
app.use('/api/v1/productos', api);

const aplicacion = require('./routes/aplicacion.route'); // Rutas dónde definimos las URL de la aplicación principal
// Ruta de la aplicación principal que apuntará a la RAIZ de la URL de nuestro dominio: /
app.use('/',aplicacion)

// Cargamos motor de plantillas
const pug = require("pug");

// Con Express no es necesario cargar el motor de plantillas,
// pero sí es necesario indicarle que lo estamos usando con:
app.set("view engine", "pug");

// Configuramos el directorio de las plantillas
app.set('views', __dirname + '/views');

//////////////// MOONGOOSE ////////////////////

// Definimos las opciones de conexión a MongoDB:
const opcionesMongo = {
    keepAlive: 1,
    useUnifiedTopology: true,
    useNewUrlParser: true,
    useFindAndModify: false
  };

var mongoose = require('mongoose');

// Ajustamos la conexión a nuestro servidor MongoDB
// mongoose.connect('mongodb://username:password@host:port/database');
// Cuando nos conectamos si hay un error en la autenticación o el servidor está caído, a los 10 segundos aparecerá el error en la co
var mongoDB = 'mongodb://tienda:abc123.@127.0.0.1:27017/tienda';

// Realizamos la conexión inicial
mongoose.connect(mongoDB,opcionesMongo).catch(error => { console.log('No me puedo conectar al servidor MongoDB: '+error)} );

// Programamos el evento de error para que se dispare si se produce algún error después de la conexión inicial con la base de datos
mongoose.connection.on('error', error => { console.log('Se ha producido un error en conexión a MongoDB: '+error)} );
mongoose.connection.on('connected', () => { console.log('Conectado a servidor MongoDB.')} );

//////////////// FIN SECCION MOONGOOSE ////////////////////
```

```
// Configuramos el puerto de escucha de nuestro servidor web.
// Asegurarse de abrir puerto en el servidor de Google, Amazon, etc...
let port = 8080;
app.listen(port, () => {
    console.log('Servidor funcionando en el puerto: ' + port);
});
```

8.11.4 Creación del fichero controlador de la aplicación

Aquí se muestra el contenido definitivo de lo que será el fichero controlador de la aplicación.

Cada vez que hagamos una nueva ruta de la aplicación, la lógica del controlador asociado a esa ruta estará aquí:

```
// aplicacion.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /pruebapug pasándole un nombre a template1.
exports.aplicacion_pruebapug = function (req, res) {
    res.render("template1",{"nombre":"Rafa"});
};


exports.productos_altas_frm = function (req, res) {
    res.render("productos_altas_frm",{"titulo":"Alta de Productos"});
};


exports.productos_bajas_frm = function (req, res) {
    // Hacemos la búsqueda de los productos en el controlador y le pasaremos la lista de productos a la vista.
    // Véase: https://mongoosejs.com/docs/api.html#model_Model.find
    var query= Producto.find({},null,{sort:{nombre:1}},function(err,docs)
    {
        res.render("productos_bajas_frm",{"titulo":"Baja de Productos","datos":docs});
    });
};


exports.productos_modificaciones_frm = function (req, res) {
    // Hacemos la búsqueda de los productos en el controlador y le pasaremos la lista de productos a la vista.
    // Véase: https://mongoosejs.com/docs/api.html#model_Model.find
    var query= Producto.find({},null,{sort:{nombre:1}},function(err,docs)
    {
        res.render("productos_modificaciones_frm",{"titulo":"Modificación de Productos","datos":docs});
    });
};

exports.productos_consultas_frm = function (req, res) {
    // Hacemos la búsqueda de los productos en el controlador y le pasaremos la lista de productos a la vista.
    // Véase: https://mongoosejs.com/docs/api.html#model_Model.find
    var query= Producto.find({},null,{sort:{nombre:1}},function(err,docs)
    {
        // console.log(docs);
        res.render("productos_consultas_frm",{"titulo":"Búsqueda de Productos","datos":docs});
    });
};
```

8.11.5 Creación de una plantilla Pug de ejemplo

Vamos a crear una plantilla de ejemplo que luego llamaremos desde una ruta determinada:

Aquí tenéis un PDF con información sobre las plantillas Pug: **https://riptutorial.com/Download/pug-es.pdf**

Esta plantilla la grabaremos con la extensión **.pug** en en directorio **views** de la aplicación:

Contenido del fichero **template1.pug** en el directorio **views**:

```
//- template1.pug
doctype 5
html(lang="es")
 head
  title Plantilla ejemplo con Pug
 body
  h1#encabezado Plantilla de ejemplo con Pug (antiguo Jade)
  p.centrar Texto de ejemplo de párrafo
  p Ejemplo de plantilla que muestra el nombre recibido en la plantilla: #{nombre}.
```

8.11.6 Creación de una ruta para probar la plantilla de ejemplo

Para probar la plantilla, crearemos una **ruta** de prueba en nuestro controlador de aplicación principal: **aplicacion.route.js**

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);

module.exports = router;
```

Programamos en el **controlador** la lógica que gestionará la ruta anterior: **aplicacion.controlller.js**

En este caso se encarga de **renderizar la vista template1** pasándole un parámetro nombre con el valor Rafa.

```
// aplicacion.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /pruebapug pasándole un nombre a template1.
exports.aplicacion_pruebapug = function (req, res) {
    res.render("template1",{"nombre":"Rafa"});
};
```

Arrancaremos de nuevo nuestro servidor de node: **node index.js**

Probaremos a conectarnos en la ruta de pruebas: **http://veiga.dynu.net:8080/pruebapug**

Vista del código fuente HTML del fichero generado en la vista:

```
<!DOCTYPE 5><html lang="es">
<head>
    <title>Plantilla ejemplo con Pug</title>
</head>
<body>
    <h1 id="encabezado">Plantilla de ejemplo con Pug (antiguo Jade)</h1>
    <p class="centrar">Texto de ejemplo de párrafo</p>
    <p>Ejemplo de plantilla que muestra el nombre recibido en la plantilla: Rafa.</p>
</body>
</html>
```

8.11.7 Creación de una plantilla esqueleto de la aplicación

Si nuestra aplicación va a tener una cierta **estructura** como una cabecera, menú, parte central, pie de página, etc... y queremos **compartirla entre todas las páginas**, lo mejor es crear una plantilla esqueleto que luego compartiremos entre el resto de plantillas.

De esta forma cada página específica solamente incluirá su código específico y no tendremos que repetir las cabeceras, pies de página, etc...

Veamos el contenido de una plantilla esqueleto a la que llamaremos **estructura.pug**:

```
//- estructura.pug
doctype html
html
  head
    meta(name='viewport' content='width=device-width, initial-scale=1, shrink-to-fit=no')
    meta(charset='utf-8')
    title= titulo
    link(rel='stylesheet', href='/css/bootstrap.min.css')

  body
    div.container
      div.row
        div.col-12
          a(href='/') Inicio
          |
          |
          a(href='/altas') Altas
          |
          |
          a(href='/bajas') Bajas
          |
          |
          a(href='/modificaciones') Modificaciones
          |
          |
          a(href='/consultas') Consultas
          |
          |
        div.col-12
          block titulo

        div.col-6
          block contenido

      block foot
        #footer
          p Aplicación de ejemplo Node.js y MongoDB. Rafa Veiga.

    block scriptscomun
      script(src='/js/jquery.min.js')
      script(src='/js/bootstrap.min.js')

    block scriptspagina
```

8.11.8 Configuración de aplicación principal index.js para trabajar con jQuery, Bootstrap y favicon

- Vamos a hacer algunas modificaciones en la aplicación principal, para poder trabajar con **ficheros estáticos** en la aplicación (funciones propias de Javascript, CSS, etc..) que se encuentran dentro de la carpeta **public**.
- Vamos a añadir también la opción de mostrar el **favicon** en la aplicación: podemos descargarnos un **favicon.ico** desde **https://www.favicon.cc/** y colocarlo en la carpeta **public**.
- También configuraremos rutas estáticas a ficheros específicos de jQuery y Bootstrap.

Para ello tendremos que instalar 2 módulos y modificaremos la configuración en la aplicación de Node, para que use esos módulos:

```
# Instalamos los siguientes módulos (solamente los módulos de producción con --production :
npm install --production --save serve-favicon path jquery bootstrap
```

Agregamos estas líneas a **index.js**:

```
// index.js

const path = require ('path');
const favicon = require('serve-favicon');
...
// Para que envíe el favicon al navegador que lo solicite. El favicon.ico estará en la carpeta public.
app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));

// Configuración para trabajar con ficheros estáticos que estén dentro de la carpeta /public
```

```
app.use(express.static(path.join(__dirname, 'public')));

// Rutas estáticas específicas adicionales, en otras carpetas distintas a /public para jQuery y Bootstrap.
app.use('/css', express.static(__dirname + '/node_modules/bootstrap/dist/css'));
app.use('/js', express.static(__dirname + '/node_modules/jquery/dist'));
app.use('/js', express.static(__dirname + '/node_modules/bootstrap/dist/js'));
...
```

Código fuente de **index.js**:

```
//index.js

const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const path = require ('path');
const favicon = require('serve-favicon');

// Parte de bodyParser, que incluiremos antes de la importación de las rutas.
app.use(bodyParser.json()); // Para parsear application/json
app.use(bodyParser.urlencoded({extended: false})); // Para parsear application/x-www-form-urlencoded

const api = require('./routes/api.route'); // Importamos las rutas de la futura API
// Le indicamos a la aplicación que al poner /api/v1/productos, dichas rutas las va a gestionar api.route.js
app.use('/api/v1/productos', api);

const aplicacion = require('./routes/aplicacion.route'); // Rutas dónde definimos las URL de la aplicación principal
// Ruta de la aplicación principal que apuntará a la RAIZ de la URL de nuestro dominio: /
app.use('/',aplicacion)

// Cargamos motor de plantillas
const pug = require("pug");

// Con Express no es necesario cargar el motor de plantillas,
// pero sí es necesario indicarle que lo estamos usando con:
app.set("view engine", "pug");

// Configuramos el directorio de las plantillas
app.set('views', __dirname + '/views');

// Para que envíe el favicon al navegador que lo solicite. El favicon.ico estará en la carpeta public.
app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));

// Configuración para trabajar con ficheros estáticos que estén dentro de la carpeta /public
app.use(express.static(path.join(__dirname, 'public')));

// Rutas estáticas específicas adicionales, en otras carpetas distintas a /public para jQuery y Bootstrap.
app.use('/css', express.static(__dirname + '/node_modules/bootstrap/dist/css'));
app.use('/js', express.static(__dirname + '/node_modules/jquery/dist'));
app.use('/js', express.static(__dirname + '/node_modules/bootstrap/dist/js'));


//////////////// MOONGOOSE ////////////////////

// Definimos las opciones de conexión a MongoDB:
const opcionesMongo = {
    keepAlive: 1,
    useUnifiedTopology: true,
    useNewUrlParser: true,
    useFindAndModify: false
  };

var mongoose = require('mongoose');

// Ajustamos la conexión a nuestro servidor MongoDB
// mongoose.connect('mongodb://username:password@host:port/database');
// Cuando nos conectamos si hay un error en la autenticación o el servidor está caído, a los 10 segundos aparecerá el error en la co
var mongoDB = 'mongodb://tienda:abc123.@127.0.0.1:27017/tienda';

// Realizamos la conexión inicial
mongoose.connect(mongoDB,opcionesMongo).catch(error => { console.log('No me puedo conectar al servidor MongoDB: '+error)} );

// Programamos el evento de error para que se dispare si se produce algún error después de la conexión inicial con la base de datos
```

```
mongoose.connection.on('error', error => { console.log('Se ha producido un error en conexión a MongoDB: '+error)} );
mongoose.connection.on('connected', () => { console.log('Conectado a servidor MongoDB.')} );

///////////////// FIN SECCION MOONGOOSE ////////////////////

// Configuramos el puerto de escucha de nuestro servidor web.
// Asegurarse de abrir puerto en el servidor de Google, Amazon, etc...
let port = 8080;
app.listen(port, () => {
    console.log('Servidor funcionando en el puerto: ' + port);
});
```

8.11.8.1 Fichero de rutas de la aplicación

Aquí tenemos las rutas que utilizaremos para la aplicación.

Creamos las ruta en **aplicacion.route.js**:

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);

// Ruta para mostrar el formulario de altas de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);

// Ruta para mostrar el formulario de bajas de productos por GET.
router.get('/bajas', aplicacion_controller.productos_bajas_frm);

// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/modificaciones', aplicacion_controller.productos_modificaciones_frm);

// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/consultas', aplicacion_controller.productos_consultas_frm);

module.exports = router;
```

8.11.8.2 ALTAS

Vamos a hacer un **formulario de altas de productos**, que almacene en MongoDB dichos productos:

1.- Primero tenemos que programar las **rutas** dentro del controlador **aplicacion.route.js**, que se encargarán de gestionar lo siguiente:

- Por un lado el **mostrar el formulario** cuando accedemos a /**altas**. Ruta de tipo **GET** en el controlador **aplicacion.route.js**.
- Aprovechamos que tenemos la API REST programado y entonces la ruta POST que recibirá los datos del formulario de altas será: /api/v1/productos/create (programado previamente en **api.controller.js**.

Añadimos al final del fichero aplicacion.route.js las siguientes líneas:

```
// aplicacion.route.js

....
// Ruta para mostrar el formulario de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);
```

Contenido actual del fichero **aplicacion.route.js**:

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);

// Ruta para mostrar el formulario de altas de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);

module.exports = router;
```

2.- A continuación haremos la **plantilla que nos mostrará la vista** con el formulario de altas:

- Dentro de la carpeta **views** haremos un nuevo fichero llamado **productos_altas_frm.pug**, con el siguiente contenido.
- En el fichero **productos_altas_frm.pug** tendremos que **extender la plantilla estructura.pug** con extends estructura.pug.
- Los bloques que no redefinamos se copiarán de la plantilla principal en la posición indicada.
- Fijarse que al final de la plantilla estamos cargando un fichero javascript dónde programaremos nuestras peticiones Ajax a la API REST.

Veamos el contenido del fichero **productos_altas_frm.pug**:

```
//- productos_altas_frm.pug
extends estructura.pug

block titulo
  //- Este comentario no se muestra en la vista resultante
  h1= titulo

block contenido
  form#formulario
     div.form-group
      label(for='name') Nombre producto:
      input#nombre.form-control(type='text',placeholder='Nombre del producto' name='nombre' required='required')

     div.form-group
      label(for='name') Precio:
      input#precio.form-control(type='number',name='precio' required='required' step="any")

     div.form-group
      button.btn.btn-secondary(type="reset") Limpiar
      button.btn.btn-primary(type="submit") Dar de Alta

  div#resultado

block scriptspagina
  script(src='/js/productos_altas_frm.js')
```

3.- Contenido del fichero /**public/js/productos_altas_frm.js**:

Aquí programaremos el código JavaScript de jQuery dónde haremos la petición a la API y gestionaremos el resultado:

```
$(function()
{
  $("#formulario").submit(function(evento)
    {
        evento.preventDefault();
        $.post("/api/v1/productos/create",{nombre:$("#nombre").val(),precio:$("#precio").val()},function(resultado)
        {
            if (resultado.status=='ok')
            {
                $("#resultado").html('').hide().addClass("alert alert-success").attr("role","alert").html(resultado.data).fadeIn(100
                $("#nombre,#precio").val('');
```

```
                }
            });
        });

    });
```

4.- Aspecto del alta realizada con Ajax y ulizando la API REST programada anteriormente.

Nos conectamos a la ruta: **http://veiga.dynu.net:8080/altas** para probar el alta:

8.11.8.3 BAJAS

- Para programar las bajas, mostraremos un formulario con un **desplegable** dónde seleccionaremos el producto que queremos dar de baja y un **botón** para dar de baja.
- En el momento que se produce la baja mostraremos un mensaje de borrado correctamente y eliminaremos de la lista el producto dado de baja.

1.- Primero tenemos que programar las **rutas** dentro del controlador **aplicacion.route.js**, que se encargarán de gestionar lo siguiente:

- Por un lado el **mostrar el formulario** cuando accedemos a /**bajas**. Ruta de tipo **GET**. (controlador **aplicacion.route.js**)

- El controlador como además se encarga de la lógica de la aplicación, se encargará de conseguir los datos que el formulario necesita para cubrir el desplegable.
- Esos datos se le pasarán a la vista, la cuál se encargará de cubrir todas las options en el desplegable de tipo select.
- Aprovechamos que tenemos la API REST programada para gestionar el borrado, y entonces la ruta que recibirá los datos del formulario de bajas será de tipo DELETE a pasando el id a borrar: /api/v1/productos/id (programado en el controlador **api.route.js**).

Añadimos al final del fichero **aplicacion.route.js** las siguientes líneas:

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);

// Ruta para mostrar el formulario de altas de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);

// Ruta para mostrar el formulario de bajas de productos por GET.
router.get('/bajas', aplicacion_controller.productos_bajas_frm);

module.exports = router;
```

2.- A continuación haremos la **plantilla que nos mostrará la vista** con el formulario de bajas:

- Dentro de la carpeta **views** haremos un nuevo fichero llamado **productos_bajas_frm.pug**, con el siguiente contenido.
- En el fichero **productos_bajas_frm.pug** tendremos que **extender la plantilla estructura.pug** con extends estructura.pug.
- Los bloques que no redefinamos se copiarán de la plantilla principal en la posición indicada.
- Fijarse que al final de la plantilla estamos cargando un fichero javascript dónde programaremos nuestras peticiones Ajax a la API REST.

Veamos el contenido del fichero **productos_bajas_frm.pug**:

```
//- productos_bajas_frm.pug
extends estructura.pug

block titulo
  //- Este comentario no se muestra en la vista resultante
  h1= titulo

block contenido

  form#formulario
     div.form-group
      label(for='name') Seleccione producto:
      select#id.form-control(name='id')
        each valor in datos
          option(value=valor._id) Producto: #{valor.nombre} al precio de: #{valor.precio}?

     div.form-group
      button.btn.btn-secondary(type="reset") Limpiar
      button.btn.btn-primary(type="submit") Dar de Baja

  div#resultado

block scriptspagina
  script(src='/js/productos_bajas_frm.js')
```

3.- Contenido del fichero /**public**/**js**/**productos_bajas_frm.js**:

Aquí programaremos el código JavaScript de jQuery dónde haremos la petición a la API y gestionaremos el resultado:

```
$(function()
{
    $("#formulario").submit(function(evento)
    {
        evento.preventDefault();

        // Debido a que no tenemos un método rápido como $.post para hacer una petición de tipo DELETE
        // tenemos que usar $.ajax para realizar dicha petición:

        var id_seleccionado=$("#id option:selected").val();

        $.ajax({
            url: "/api/v1/productos/"+id_seleccionado,
            type: 'DELETE',
            success: function(data,textStatus,request) {
                if (request.status==204)
                {
                    $("#id option:selected").remove();
                    $("#resultado").html('').hide().addClass("alert alert-success").attr("role","alert").html("Producto borrado corr
                }
            },
            error: function (xhr,ajaxOptions,error)
            {
                if (xhr.status==404)
                {
                    $("#resultado").html('').hide().addClass("alert alert-danger").attr("role","alert").html('No se ha encontrado el
                }
            }
        });
    });
});
```

4.- Aspecto del alta realizada con Ajax y ulizando la API REST programada anteriormente.

Nos conectamos a la ruta: **http://veiga.dynu.net:8080/bajas** para probar la baja de un producto:

Baja de Productos

Seleccione producto:

Producto: Ariel 3 en 1 al precio de: 30.89€

Producto: Ariel 3 en 1 al precio de: 30.89€
Producto: ColaCao Noir al precio de: 2.35€
Producto: Detergente Colón al precio de: 5.8€
Producto: Don Limpio al precio de: 2.65€
Producto: Finish Powerball al precio de: 15.6€
Producto: KH-7 al precio de: 2.68€
Producto: Samsung Galaxy S5 al precio de: 340€
Producto: Samsung Galaxy S5 al precio de: 340€
Producto: Tambor Detergente Skip al precio de: 6.9€
Producto: Tenn Limpiador al precio de: 1.79€
Producto: Vileda Fregona al precio de: 3.9€

8.11.8.4 MODIFICACIONES

- Para programar las modificaciones, mostraremos un formulario con un **desplegable** dónde seleccionaremos el producto que queremos modificar y un **botón** para mostrar los datos a modificar.
- Mostraremos los antiguos datos del producto en el formulario y un botón de Modificar, que realizará el proceso de modificar en la base de datos.
- A continuación se mostrará el mensaje modificado correctamente.

1.- Primero tenemos que programar las **rutas** dentro del controlador **aplicacion.route.js**, que se encargarán de gestionar lo siguiente:

- Por un lado el **mostrar el formulario** cuando accedemos a /**modificaciones**. Ruta de tipo **GET**.
- El controlador como además se encarga de la lógica de la aplicación, se encargará de conseguir los datos que el formulario necesita para cubrir el desplegable.
- Esos datos se le pasarán a la vista, la cuál se encargará de cubrir todas las options en el desplegable de tipo select.
- Al seleccionar un prodcutos mediante jQuery mostraremos el formulario oculto con los datos del producto descargados de la base de datos.
- Aprovechamos que tenemos la API REST programada para gestionar la modificación, y entonces la ruta que recibirá los datos del formulario de bajas será de tipo PUT pasando el id a modificar: /api/v1/productos/id (programado en controlador **api.route.js**)

Añadimos al final del fichero **aplicacion.route.js** las siguientes líneas:

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);
```

```
// Ruta para mostrar el formulario de altas de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);

// Ruta para mostrar el formulario de bajas de productos por GET.
router.get('/bajas', aplicacion_controller.productos_bajas_frm);

// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/modificaciones', aplicacion_controller.productos_modificaciones_frm);

module.exports = router;
```

2.- A continuación haremos la **plantilla que nos mostrará la vista** con el formulario de modificaciones:

- Dentro de la carpeta **views** haremos un nuevo fichero llamado **productos_modificaciones_frm.pug**, con el siguiente contenido.
- En el fichero **productos_modificaciones_frm.pug** tendremos que **extender la plantilla estructura.pug** con extends estructura.pug.
- Los bloques que no redefinamos se copiarán de la plantilla principal en la posición indicada.
- Fijarse que al final de la plantilla estamos cargando un fichero javascript dónde programaremos nuestras peticiones Ajax a la API REST.

Veamos el contenido del fichero **productos_modificaciones_frm.pug**:

```
//- productos_modificaciones_frm.pug
extends estructura.pug

block titulo
  //- Este comentario no se muestra en la vista resultante
  h1= titulo

block contenido

  form#formModif1
     div.form-group
      label(for='name') Seleccione producto a modificar:
      select#id.form-control(name='id')
        each valor in datos
          option(value=valor._id) Producto: #{valor.nombre} al precio de: #{valor.precio}?

     div.form-group
      button.btn.btn-secondary(type="reset") Limpiar
      button.btn.btn-primary(type="submit") Editar datos

  form#formModif2
    div.form-group
      label(for='name') Nombre producto:
      input#nombre.form-control(type='text',placeholder='Nombre del producto' name='nombre' required='required')

    div.form-group
      label(for='name') Precio:
      input#precio.form-control(type='number',name='precio' required='required' step="any")

      input#_id.form-control(type='hidden',name='_id')

    div.form-group
      button.btn.btn-secondary(type="reset") Limpiar
      button.btn.btn-primary(type="submit") Modificar Datos

  div#resultado

block scriptspagina
  script(src='/js/productos_modificaciones_frm.js')
```

3.- Contenido del fichero **views/estructura.pug**: Hemos modificado también el fichero estructura.pug para incluir una **hoja de estilos misestilos.css** dónde ocultamos por defecto el formulario2 de edición de datos.

```
//- estructura.pug
doctype html
html
  head
    meta(name='viewport' content='width=device-width, initial-scale=1, shrink-to-fit=no')
```

```
    meta(charset='utf-8')
    title= titulo
    link(rel='stylesheet', href='/css/misestilos.css')
    link(rel='stylesheet', href='/css/bootstrap.min.css')

  body
    div.container
      div.row
        div.col-12
          a(href='/') Inicio
          |
          |
          a(href='/altas') Altas
          |
          |
          a(href='/bajas') Bajas
          |
          |
          a(href='/modificaciones') Modificaciones
          |
          |
          a(href='/consultas') Consultas
          |
          |
        div.col-12
          block titulo

        div.col-6
          block contenido

      block foot
        #footer
          p Aplicación de ejemplo Node.js y MongoDB. Rafa Veiga.

    block scriptscomun
      script(src='/js/jquery.min.js')
      script(src='/js/bootstrap.min.js')

    block scriptspagina
```

Contenido de la hoja de estilos misestilos.css:

```
form#formModif2
{
    visibility: hidden;
}
```

4.- Contenido del fichero /**public/js/productos_modificaciones_frm.js**:

Aquí programaremos el código JavaScript de jQuery dónde haremos la petición a la API y gestionaremos el resultado:

```
$(function()
{
    $("#formModif1").submit(function(evento)
    {
        evento.preventDefault();

        // Vamos a usar el método $.ajax por que aunque podríamos usar $.get, no tenemos acceso al código 404 cuando no encuentra un

        var id_seleccionado=$("#id option:selected").val();

        $.ajax({
            url: "/api/v1/productos/"+id_seleccionado,
            type: 'GET',
            success: function(respuesta,textStatus,request) {
                if (request.status==200)
                {
                   $('#formModif1').hide();
                   $("#formModif2 #nombre").val(respuesta.data.nombre);
                   $("#formModif2 #precio").val(respuesta.data.precio);
                   $("#formModif2 #_id").val(respuesta.data._id);
```

```
                $('#formModif2').css('visibility', 'visible');
            }
        },
        error: function (xhr,ajaxOptions,error)
        {
         if (xhr.status==404)
         {
             $("#resultado").html('').hide().addClass("alert alert-danger").attr("role","alert").html('No se ha encontrado el pro
         }
    }
});
});



$("#formModif2").submit(function(evento)
{
    evento.preventDefault();

    // Vamos a usar el método $.ajax por que aunque podríamos usar $.get, no tenemos acceso al código 404 cuando no encuentra un

    var id_seleccionado=$("#formModif2 #_id").val();

    $.ajax({
        url: "/api/v1/productos/"+id_seleccionado,
        type: 'PUT',
        data: {"nombre": $("#formModif2 #nombre").val(), "precio":$("#formModif2 #precio").val()},
        success: function(respuesta,textStatus,request) {
            if (request.status=='200')
            {
                $("#formModif2").hide();
                $("#resultado").html('').hide().addClass("alert alert-success").attr("role","alert").html('Se han actualizado lo
            }
        },
        error: function (xhr,ajaxOptions,error)
        {
            if (xhr.status==404)
            {
                $("#resultado").html('').hide().addClass("alert alert-danger").attr("role","alert").html('No se ha encontrado el
            }
        }
    });
});

});
```

5.- Aspecto de una baja realizada con Ajax y ulizando la API REST programada anteriormente.

Nos conectamos a la ruta: **http://veiga.dynu.net:8080/modificaciones** para probar la modificación de un producto:

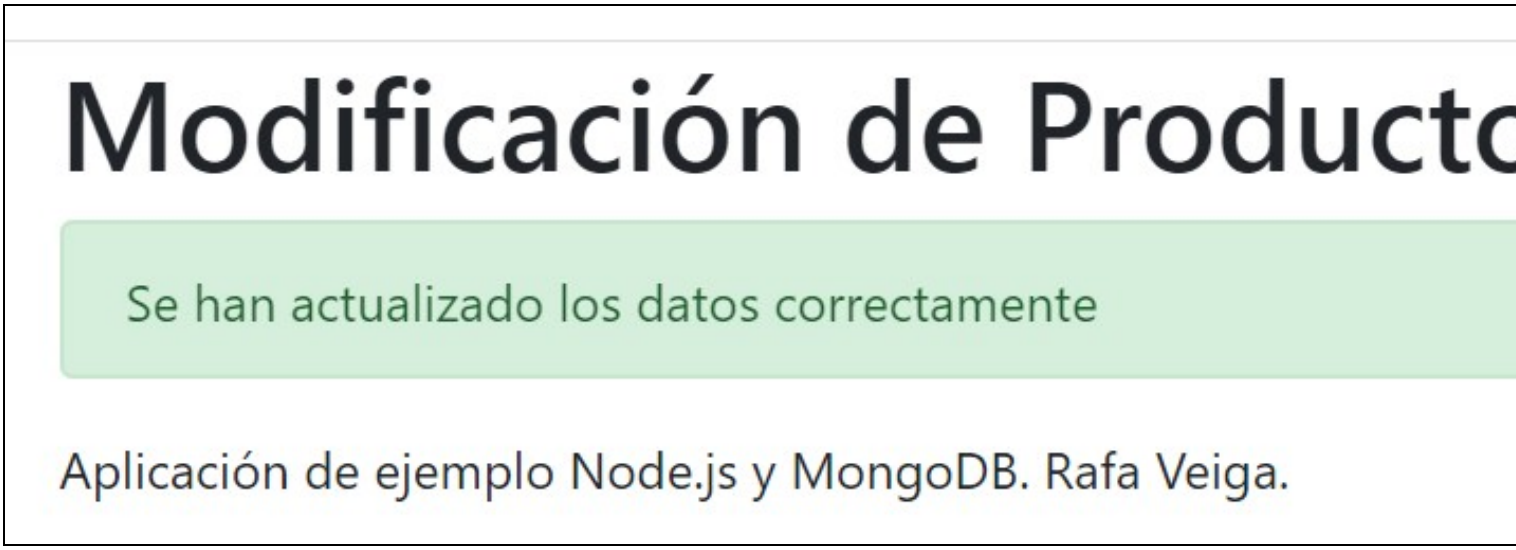
Modificación de Producto

Seleccione producto a modificar:

Producto: Ariel 3 en 2 al precio de: 31.89€

Limpiar Editar datos

Aplicación de ejemplo Node.js y MongoDB. Rafa Veiga.



8.11.8.5 CONSULTAS

Vamos a implementar la página de consultas en la que a través de un formulario podamos hacer búsquedas por nombre o precio.

1.- Primero tenemos que programar las **rutas** dentro del controlador **aplicacion.route.js**, que se encargarán de gestionar lo siguiente:

- Por un lado el **mostrar el formulario** cuando accedemos a /**consultas**. Ruta de tipo **GET** en el controlador **aplicacion.route.js**.
- Aprovechamos que tenemos la API REST programado y entonces la ruta POST que recibirá los datos del formulario de altas será: /api/v1/productos/create (programado previamente en **api.controller.js**. ??

Añadimos al final del fichero aplicacion.route.js las siguientes líneas:

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);

// Ruta para mostrar el formulario de altas de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);

// Ruta para mostrar el formulario de bajas de productos por GET.
router.get('/bajas', aplicacion_controller.productos_bajas_frm);

// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/modificaciones', aplicacion_controller.productos_modificaciones_frm);

// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/consultas', aplicacion_controller.productos_consultas_frm);

module.exports = router;
```

2.- A continuación haremos la **plantilla que nos mostrará la vista** con el formulario de consultas:

- Dentro de la carpeta **views** haremos un nuevo fichero llamado **productos_consultas_frm.pug**, con el siguiente contenido.
- En el fichero **productos_consultas_frm.pug** tendremos que **extender la plantilla estructura.pug** con extends estructura.pug.
- Los bloques que no redefinamos se copiarán de la plantilla principal en la posición indicada.
- Fijarse que al final de la plantilla estamos cargando un fichero javascript dónde programaremos nuestras peticiones Ajax a la API REST.

Veamos el contenido del fichero **productos_consultas_frm.pug**:

```
//- productos_consultas_frm.pug
extends estructura.pug

block titulo
  //- Este comentario no se muestra en la vista resultante
  h1= titulo

block contenido
  form#formulario
     div.form-group
      label(for='name') Texto a buscar:
      input#busqueda.form-control(type='text',placeholder='Productos cuyo nombre contenga el siguiente texto. Introduzca aquí.' name=

     div.form-group
      button.btn.btn-secondary(type="reset") Limpiar
      button.btn.btn-primary(type="submit") Buscar productos

  div#resultado

block scriptspagina
  script(src='/js/productos_consultas_frm.js')
```

3.- Modificación de la API REST de consultas para permitir este tipo de búsqueda: En el fichero **api.controller.js** hemos modificado la función exports.api_productos_read para que permita realizar búsquedas por parámetros, quedando el contenido del fichero definitivo así:

```
// api.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /api/v1/productos/test
```

```
exports.api_test = function (req, res) {
    res.send('Saludos desde el controlador de pruebas /api/v1/productos/test !');
};


exports.api_productos_create = function (req, res) {
    let producto = new Producto(
    {
        nombre: req.body.nombre,
        precio: req.body.precio
    }
    );

    producto.save(function (err) {
        if (err)
        {
            // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
            return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!"
        }

        // Enviamos al cliente la siguiente respuesta con el código HTTP 200.
        res.type('json').status(200).send({ status: "ok", data: "Producto creado satisfactoriamente!" });
    })
};


exports.api_productos_read = function (req, res) {
    // Si a esta ruta le pasamos una cadena de consulta (query string) del estilo ?busqueda=xxxxx realiza la búsqueda de productos c
    // Con req.query tenemos acceso a las variables recibidas por GET.
    if (req.query.busqueda) // Si ?busqueda contiene algún texto.
    {
        // Referencia para este tipo de búsqueda en:
        // https://stackoverflow.com/questions/8246019/case-insensitive-search-in-mongo
       Producto.find({nombre:{'$regex' : req.query.busqueda, '$options' : 'i'}}, function (err, productos) {
        if (productos.length==0)
        {
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado ningún producto cuyo nombre contenga: "+req.query.busc
        }
        else
        {
            // También podemos devolver así la información:
            res.status(200).json({ status: "ok", total:productos.length, data: productos });
        }
    })
   }
   else if (req.params.id)
   {
     // Con req.params tenemos acceso a los parámetros extraídos de la ruta.
     // Si le pasamos un ID de producto devuelve los datos de ese producto
     Producto.findById(req.params.id, function (err, unProducto) {
        if (err)
        {
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
        }
        else
        {
            // También podemos devolver así la información:
            res.status(200).json({ status: "ok", data: unProducto });
        }
    })
 }
 else
 {
   // En otro caso devolverá todos los productos
   // Ayuda: https://mongoosejs.com/docs/api/model.html
   Producto.find({}, function (err, todosProductos) {
    if (err)
          // Si se ha producido un error, salimos de la función devolviendo  código http 422 (Unprocessable Entity).
          return (res.type('json').status(422).send({ status: "error", data: "No se puede procesar la entidad, datos incorrectos!" })

    // También podemos devolver así la información:
```

```
    res.status(200).json({ status: "ok", data: todosProductos });
})
}
};


exports.api_productos_update = function (req, res) {
    Producto.findByIdAndUpdate(req.params.id, { $set: req.body }, function (err, productoActualizado) {
        if (err)
        {
            //res.send(err);
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
        }
        else
        {
            // Devolvemos el código HTTP 200.
            res.status(200).json({ status: "ok", data: productoActualizado });
        }
    });
};


exports.api_productos_delete = function (req, res) {
    Producto.findByIdAndRemove(req.params.id, function(err, data) {
        if (err || !data) {
            //res.send(err);
            // Devolvemos el código HTTP 404, de producto no encontrado por su id.
            res.status(404).json({ status: "error", data: "No se ha encontrado el producto con id: "+req.params.id});
        }
        else
        {
            // Devolvemos el código HTTP 204 cuando lo ha borrado (No Content). No devuelve contenido.
            res.status(204).json({ status: "ok", data: "Se ha eliminado correctamente el producto con id: "+req.params.id});
        }
    });
};
```

4.- Contenido del fichero /**public/js/productos_consultas_frm.js**:

Aquí programaremos el código JavaScript de jQuery dónde haremos la petición a la API y gestionaremos el resultado:

```
$(function()
{
    $("#formulario").submit(function(evento)
    {
        evento.preventDefault();

        var texto_busqueda = $("#formulario #busqueda").val();

        // Limpiamos el div de resultados.
        $("#resultado").html('').removeClass("alert alert-danger alert-success");

        // Petición AJAX a la API.
        $.ajax({
            url: "/api/v1/productos/?busqueda="+texto_busqueda,
            type: 'GET',
            success: function(datos,textStatus,request) {
                if (request.status==200)
                {
                    var cadena='Se han encontrado <strong>'+datos.total+'</strong> productos conteniendo el texto: <strong>'+texto_bu

                    for (i=0;i<datos.total;i++)
                    {
                        cadena+= "<li> Producto: <b>" +datos.data[i].nombre+ "</b> al precio de: <b>"+datos.data[i].precio+" ?</b>.</
                    }

                    $("#resultado").html('').hide().addClass("alert alert-success").attr("role","alert").html(cadena).fadeIn(1000);
                }
            },
            error: function (xhr,ajaxOptions,error)
```

```
                {
                    if (xhr.status==404)
                    {
                        $("#resultado").html('').hide().addClass("alert alert-danger").attr("role","alert").html('No se ha encontrado ni
                    }
                }
            });
        });

    });
```

5.- Aspecto las consultas realizadas con Ajax y ulizando la API REST programada anteriormente.

Nos conectamos a la ruta: **http://veiga.dynu.net:8080/consultas** para probar el alta:

Búsqueda de Productos

Texto a buscar:

rvc

Limpiar Buscar productos

No se ha encontrado ningún producto con ese texto!

Aplicación de ejemplo Node.js y MongoDB. Rafa Veiga.

Búsqueda de Productos

Texto a buscar:

tergen

Limpiar Buscar productos

Se han encontrado **2** productos conteniendo el texto: **tergen**

- Producto: **Tambor Detergente Skip** al precio de: **6.9 €**.
- Producto: **Detergente Colón** al precio de: **5.8 €**.

Aplicación de ejemplo Node.js y MongoDB. Rafa Veiga.

8.11.9 Ruta a la página de Inicio de la aplicación

Por último nos falta crear una ruta al Inicio que muestre una página por defecto:

Creamos la ruta en **aplicacion.route.js**:

```
// aplicacion.route.js

// Cargamos el módulo express (servidor web).
const express = require('express');

// Cargamos el módulo Router
const router = express.Router();

// Cargamos el fichero de controladores (dónde se define la lógica que sucede cuando se entra en una ruta dentro de /api)
const aplicacion_controller = require('../controllers/aplicacion.controller');

// Ruta de prueba de plantillas Pug en nuestra aplicación principal:
router.get('/pruebapug', aplicacion_controller.aplicacion_pruebapug);

// Ruta para mostrar el formulario de altas de productos por GET.
router.get('/altas', aplicacion_controller.productos_altas_frm);

// Ruta para mostrar el formulario de bajas de productos por GET.
router.get('/bajas', aplicacion_controller.productos_bajas_frm);
```

```
// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/modificaciones', aplicacion_controller.productos_modificaciones_frm);

// Ruta para mostrar el formulario de modificaciones de productos por GET.
router.get('/consultas', aplicacion_controller.productos_consultas_frm);

// Ruta para mostrar la página de entrada a la aplicación.
router.get('/', aplicacion_controller.productos_inicio);

module.exports = router;
```

Creamos la lógica del controlador en **aplicacion.controller.js**:

```
// aplicacion.controller.js

// Cargamos el schema del modelo de datos para poder usarlo dentro de la lógica del controlador
// y así poder añadir, borrar, etc. sobre productos.
const Producto = require('../models/productos.model');

// Prueba a una ruta /pruebapug pasándole un nombre a template1.
exports.aplicacion_pruebapug = function (req, res) {
    res.render("template1",{"nombre":"Rafa"});
};


exports.productos_altas_frm = function (req, res) {
    res.render("productos_altas_frm",{"titulo":"Alta de Productos"});
};


exports.productos_bajas_frm = function (req, res) {
    // Hacemos la búsqueda de los productos en el controlador y le pasaremos la lista de productos a la vista.
    // Véase: https://mongoosejs.com/docs/api.html#model_Model.find
    var query= Producto.find({},null,{sort:{nombre:1}},function(err,docs)
    {
        res.render("productos_bajas_frm",{"titulo":"Baja de Productos","datos":docs});
    });
};


exports.productos_modificaciones_frm = function (req, res) {
    // Hacemos la búsqueda de los productos en el controlador y le pasaremos la lista de productos a la vista.
    // Véase: https://mongoosejs.com/docs/api.html#model_Model.find
    var query= Producto.find({},null,{sort:{nombre:1}},function(err,docs)
    {
        res.render("productos_modificaciones_frm",{"titulo":"Modificación de Productos","datos":docs});
    });
};

exports.productos_consultas_frm = function (req, res) {
    // Hacemos la búsqueda de los productos en el controlador y le pasaremos la lista de productos a la vista.
    // Véase: https://mongoosejs.com/docs/api.html#model_Model.find
    var query= Producto.find({},null,{sort:{nombre:1}},function(err,docs)
    {
        // console.log(docs);
        res.render("productos_consultas_frm",{"titulo":"Búsqueda de Productos","datos":docs});
    });
};


// Página de Inicio a la aplicación /:
exports.productos_inicio = function (req, res) {
    res.render("productos_inicio",{"titulo":"Gestión de Productos con Node.js y MongoDB"});
};
```

Por último crearemos la vista **productos_inicio.pug** de la aplicación:

```
//- productos_inicio.pug
extends estructura.pug

block titulo
```

```
    //- Este comentario no se muestra en la vista resultante
    h1= titulo

  block contenido
    p Aplicación de gestión realizada con Node.js y MongoDB como introducción a estas tecnologías de programación en el lado del servid
    p Más información y tutoriales de programación en
    a(href='https://manuais.iessanclemente.net') Manuais IES San Clemente
```

8.11.10 Descarga de la aplicación completa

Puede descargar la Archivo:TiendaApp.zip.

Veiga (discusión) 12:07 19 feb 2020 (CET)

9 Buenas prácticas de rendimiento. Mejora de Express.js en entorno de producción

https://sematext.com/blog/expressjs-best-practices/

Veiga (discusión) 21:25 19 abr 2020 (CEST)