SOD

PRÁTICA 3

SD

P3. APLICACIÓN DISTRIBUIDA CON NODEJS

PREÁMBULO

El desarrollo de una **aplicación distribuida** profesional, como por ejemplo las actuales *aplicaciones Web*, involucra multitud de tecnologías, herramientas, *frameworks*, perfiles profesionales tanto para su diseño e implementación, como para su posterior mantenimiento. Para facilitar el trabajo, el diseño y el desarrollo de este tipo de aplicaciones se suelen dividir en *front-end* (determina y proporciona la interfaz con la que el usuario interactuará con la aplicación o servicio) y en *back-end* (aborda la lógica de negocio de la aplicación en forma de *servicios desacoplados* sobre *sistemas distribuidos*).

Debido a la cantidad de herramientas que intervienen para proporcionar cobertura a cada tarea y a cada componente, la cantidad de alternativas para cada una de ellas, y la gran variedad de tecnologías y herramientas nuevas que aparecen cada día, es importante centrarse en un conjunto (una *pila* o *stack*) de tecnologías y herramientas que facilite el desarrollo de la aplicación, cubriendo todas las necesidades, trabajando de forma conjunta en un proyecto para crear una *aplicación Web*. Es por esta razón que a este tipo de desarrollos y a sus desarrolladores se les suele denominar ***Full Stack***.

En esta práctica nos centraremos en una *pila* basada en *javascript* conocida como ***pila MEAN*** (*MEAN Stack*). Su nombre proviene de las principales herramientas y tecnologías que la conforman (**figura 1**): **Mongodb**, **Express**, **Angular** y **Nodejs**.

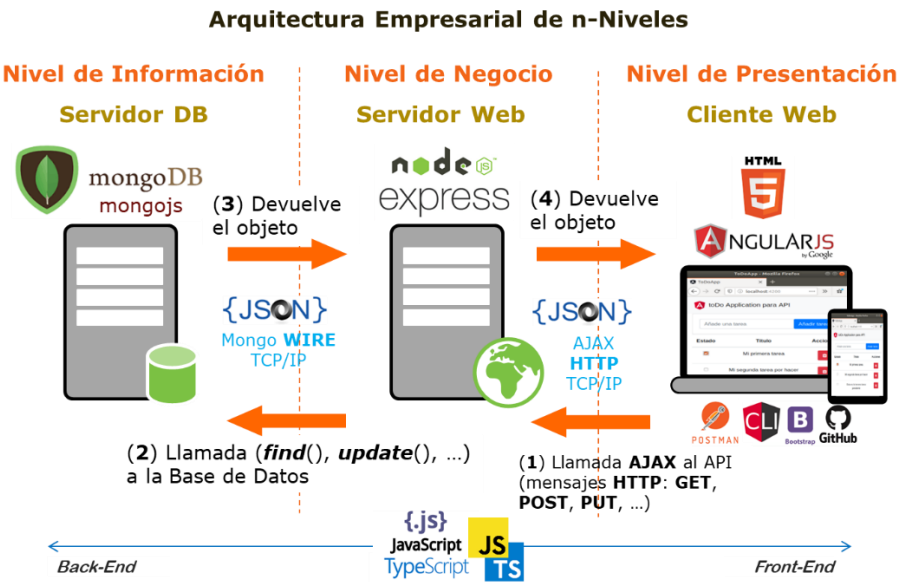


Figura 1. Tecnologías involucradas en un desarrollo basado en la **pila MEAN,** junto con la relación entre los diferentes componentes dentro de una **Arquitectura Empresarial de n-Niveles**.

En la *figura 1* se puede observar cómo se relacionan y organizan las diferentes herramientas y tecnologías que forman parte de la **Pila MEAN,** cuáles son los principales elementos que intervienen en la misma y cómo encaja dicha tecnología y elementos dentro de un estilo arquitectural basado en una **arquitectura empresarial de n-Niveles**.

Un *desarrollador Full Stack* debe ser capaz de desarrollar *front-end* y *back-end*, tener conocimientos de bases de datos, trabajar con diferentes sistemas operativos y lenguajes de programación, saber cómo diseñar una aplicación UX/UI. En definitiva, se trata de uno de los perfiles más completos y, por lo tanto, deseados y demandados en la actualidad.

En esta asignatura, debido a sus características, aunque se abordará el diseño completo de una *Aplicación Web* a través de un proyecto, nos centraremos fundamentalmente en el desarrollo del *back-end*, concretamente en el desarrollo de un *servicios* de *back-end*.

OBJETIVO DE LAS PRÁCTICAS

El objetivo global de las prácticas consiste en explorar un conjunto de tecnologías que faciliten el diseño y la realización de **aplicaciones distribuidas** basadas en *servicios distribuidos* sobre Internet.

En concreto, el principal objetivo será crear un *Servicio Web* (**WS**) que proporcione una **API RESTful** sobre HTTP que se comporte como un **CRUD** (*Create, Read, Update and Delete*) de una base de datos. Para ello se emplearán tecnologías **MEAN**: **NodeJS** para el back-end, **Express** como *framework*, **MongoDB** (junto con la biblioteca **mongoose**, para facilitar la escritura del código) como gestor de DB.

A partir de los servicios de *back-end*, como un objetivo más secundario, abordaremos el desarrollo del *front-end* que no utilizará Angular por simplicidad, sino que será una página en html+JS servida por un APACHE que permita mostrar los elementos de la Base de Datos.

REQUISITOS

Para realizar las prácticas, se debe contar con los siguientes elementos:

- Equipo con Windows, Linux u OS X con conexión a Internet.
- Tener instaladas las siguientes aplicaciones:
  - Visual Studio Code + Plugins para Node, JS y TS.
  - Postman.

    - NodeJS (node y npm).
    - Git (y una cuenta con un repositorio en GitHub o Bitbacket, por ejemplo) o smartgit
    - MongoDB.
- Conocimientos básicos de administración de SO, JS, HTML y CSS.

RECURSOS Y REFERENCIAS ADICIONALES

Node.js
https://juanda.gitbooks.io/webapps/content/api/

NodeJS. Javascript en servidor
https://juanda.gitbooks.io/webapps/content/javascript/node.html

API y Arquitectura REST (AJAX, CORS, Autenticación)
https://juanda.gitbooks.io/webapps/content/api/arquitectura-api-rest.html

Creación de un API con NodeJS
https://juanda.gitbooks.io/webapps/content/api/creacion_de_una_api_con_nodejs.html

NodeJS y SSL/TLS
https://www.sitepoint.com/how-to-use-ssltls-with-node-js/
https://blog.mgechev.com/2014/02/19/create-https-tls-ssl-application-with-express-nodejs/

Registro, Autenticación y Autorización
https://blog.restcase.com/4-most-used-rest-api-authentication-methods/

JSON Web Token (JWT)
https://jwt.io/
https://www.npmjs.com/package/jwt-simple
https://www.npmjs.com/package/bcrypt

Seguridad adicional
https://expressjs.com/es/advanced/best-practice-security.html

Single Page Applications (SPA)
https://juanda.gitbooks.io/webapps/content/spa/arquitectura_de_un_spa.html

Evolución de CSS
https://juanda.gitbooks.io/webapps/content/css/evolucion.html

Proyecto de Web Básica
https://juanda.gitbooks.io/webapps/content/proyectos/web_basica.html

Web con Bootstrap y Gulp
https://juanda.gitbooks.io/webapps/content/proyectos/bootstrap_gulp.html

Desarrollo de aplicaciones Web
https://www.gitbook.com/?utm_source=legacy&utm_medium=redirect&utm_campaign=close_legacy

INSTALACIÓN Y PRUEBA DE **NODEJS**

Node.js es un entorno de ejecución para JavaScript construido sobre el motor JavaScript V8 de Google Chrome. Fue creado para permitir a los desarrolladores ejecutar JavaScript en el servidor, lo que antes era posible solo en el navegador. Node.js utiliza un modelo de E/S no bloqueante y orientado a eventos, lo que lo hace eficiente y adecuado para aplicaciones que manejan muchas operaciones de entrada y salida, como aplicaciones en tiempo real y servicios web.

Para instalar nodejs en Windows hay que descargar la última versión estable de: https://nodejs.org/en/download/prebuilt-installer y ejecutar el instalador con los parámetros por defecto.

Para instalarlo en MAC seguir las instrucciones de: https://kinsta.com/es/blog/como-instalar-node-js/

Una vez instalado comprobamos la versión instalada desde CMD de Windows: node --version

```
C:\Users\irenf>node --version
v22.11.0
```

NPM (Node Package Manager) es el gestor de paquetes predeterminado para Node.js. Es una herramienta esencial para cualquier desarrollador que trabaje con Node.js, ya que facilita la gestión de dependencias, la instalación de paquetes y la administración de scripts de desarrollo instalación de paquetes y la administración. Con la instalación de node v 22 se instala directamente el npm, así que comprobamos que se haya instalado correctamente preguntando por su versión en el CMD: npm --version

```
C:\Users\irenf>npm --version
10.9.0
```

Probamos la consola **node**, que no deja de ser un intérprete de *JavaScript*

```
$ node
> 3+4
7
> console.log("Hola a todos!");
Hola a todos!
undefined
> .exit [o <Control+C> para salir]
$
```

GIT Y BITBUCKET

La instalación de Git es bastante sencilla y puede hacerse en Windows, macOS y Linux. A continuación, se explica cómo instalar Git en windows.

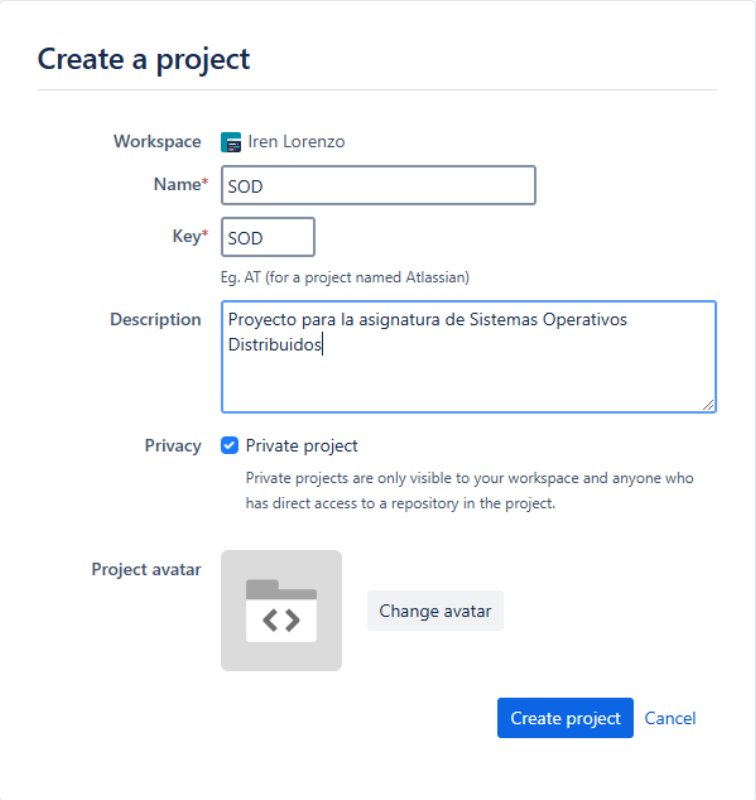
1. INSTALACIÓN DE GIT EN WINDOWS
    1. **Descargar el Instalador de Git**:
        - Ve al sitio web oficial de Git: https://git-scm.com/

- Haz clic en el botón de descarga para Windows. Esto descargará el instalador de Git para Windows.

2. **Ejecutar el Instalador**:
   - Una vez que se haya descargado el archivo .exe, haz doble clic sobre él para comenzar la instalación.
3. **Finalizar la Instalación**:
   - Haz clic en "Next" y luego en "Install". Después de la instalación, puedes hacer clic en "Finish".
4. **Verificar la Instalación**:
   - Abre el Command Prompt (CMD) para verificar que Git se instaló correctamente:
     git --version
   - Deberías ver la versión de Git instalada, por ejemplo: git version 2.x.x.windows.

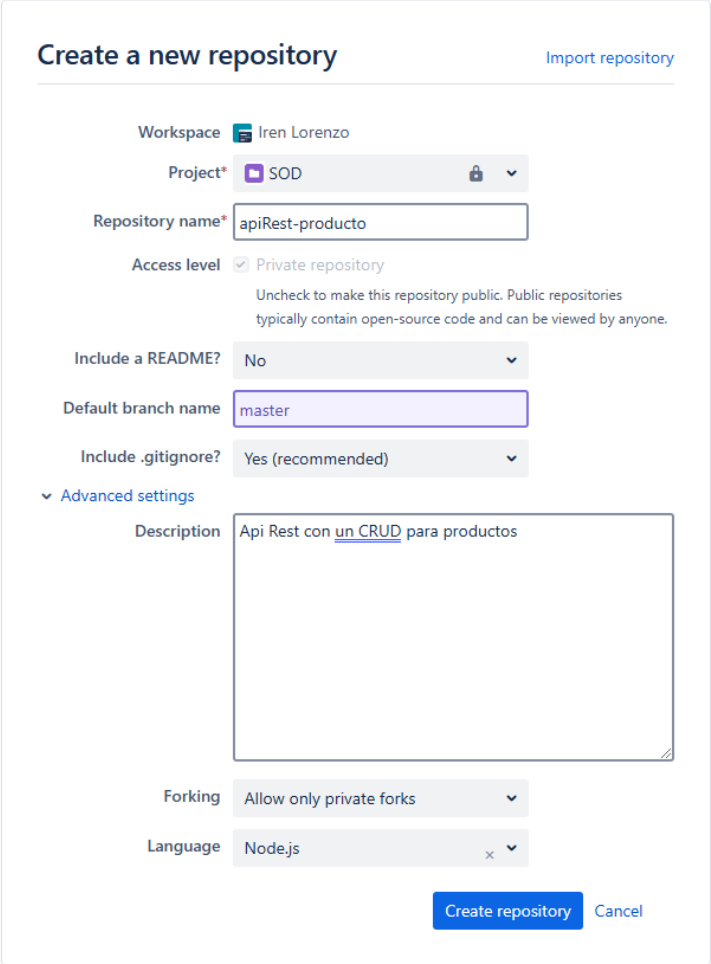
2. CREAR UN PROYECTO Y REPOSITORIO EN BITBUCKET

1. Iniciar Sesión en Bitbucket:
   - Ve a https://bitbucket.org y crea una cuenta si aún no tienes una (utilizando la cuenta de la ua).
   - Inicia sesión con tu cuenta de Bitbucket.
2. Crear un Proyecto:
   - Una vez que hayas iniciado sesión, en la barra lateral izquierda, haz clic en el icono de "Proyectos" (Project).
   - Haz clic en el botón "Crear proyecto" (Create Project).
   - Rellena los campos:
     - Nombre del Proyecto: Ingresa el nombre para el proyecto (por ejemplo, SOD).
     - Clave del Proyecto: Bitbucket generará automáticamente una clave, pero puedes cambiarla si lo prefieres.
     - Descripción: Opcionalmente, agrega una breve descripción para el proyecto.



   - Haz clic en Crear proyecto para completar la creación del proyecto.
3. Crear un Repositorio dentro del Proyecto:

- Dentro del proyecto recién creado, haz clic en el botón "Crear repositorio" (Create repository).
- Rellena los campos:
  - Project: SOD (que es el proyecto que acabamos de crear)
  - Nombre del repositorio: Elige un nombre para tu repositorio (por ejemplo, apiRest-producto).
  - Default brach name: master
  - En Advanced settings
    1. Descripción: Agrega una breve descripción de tu repositorio (opcional).
    2. Language: Nodejs.



- Haz clic en Crear repositorio.

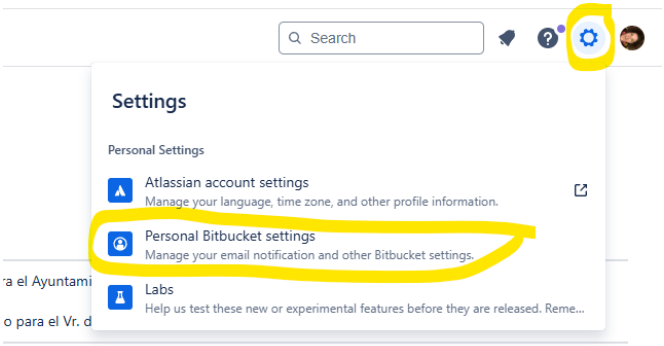
Una vez creado el repositorio, verás la página del repositorio donde puedes ver la URL remota del repositorio, instrucciones para clonar y más.

3. CONFIGURAR GIT CON AUTENTICACIÓN EN BITBUCKET

Para poder interactuar con el repositorio remoto en Bitbucket, necesitas autenticarte correctamente. Aquí tienes una opción de autenticación **con Token (PAT)**

Opción 1: Autenticación con Token (PAT)

1. **Generar un Token de Acceso Personal (PAT)**:
   - Ve a Bitbucket y en la esquina superior derecha haz clic en configuración, luego selecciona **Personal Bitbucket settings**.



- En la sección **Access Management**, selecciona **App passwords**.
- Haz clic en **Create app password**.
- Asigna un nombre al token (por ejemplo, git-pass) y selecciona los permisos que necesites, como **Read and Write** en los repositorios.
- Haz clic en **Create**. Guarda el **App password** generado, ya que no podrás verlo de nuevo.

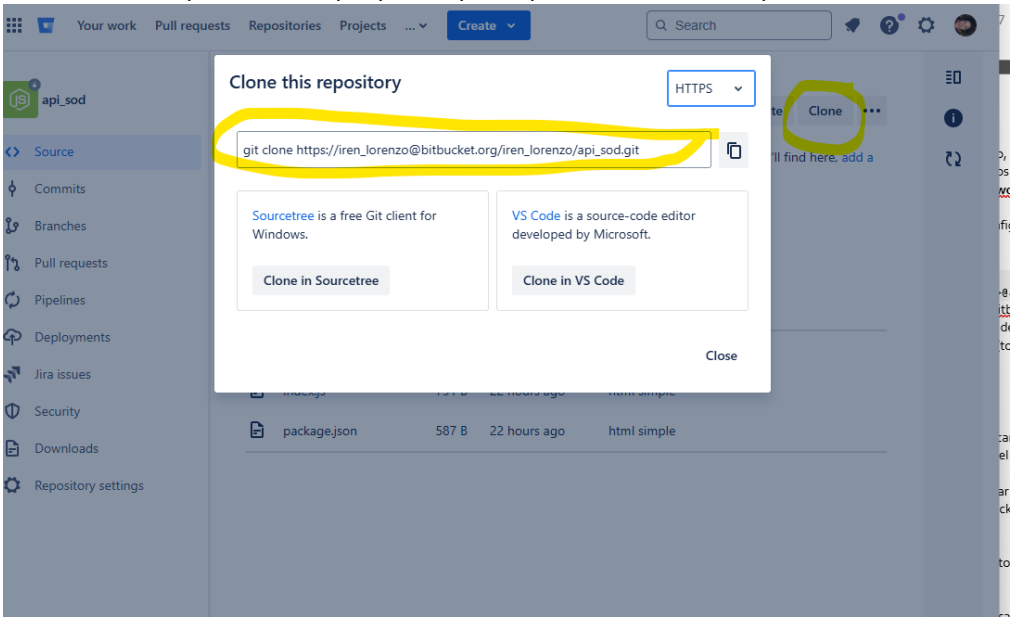
2. **Configurar Git para Usar el Token**:
   - Abre Git Bash o la terminal (CMD) y configura tu nombre y correo de usuario para que estén asociados con tus commits:

```
$ git config --global user.name <user>
$ git config --global user.email <user>@alu.ua.es
```

   - Cuando realices un git push o git pull, Bitbucket te pedirá las credenciales. Usa:
     - **Usuario**: Tu nombre de usuario de Bitbucket.
     - **Contraseña**: El **App password** (token) generado anteriormente.

4. Clonar el Repositorio a una Carpeta Local

1. **Clonar el Repositorio**:
   - Abre tu terminal CMD en la carpeta donde quieres clonar el proyecto localmente.
   - Ve a Bitbucket y dentro del proyecto, y el repositorio creado copia el comando de clonación:



   - Ejecuta el comando copiado en la consola abierta en la carpeta selecciona del paso 1.

2. **Verificar el Clonado**:
   - Navega dentro de la carpeta del proyecto:

cd apiRest-producto
git status

Deberías ver que el repositorio está limpio y listo para usarse.

CREACIÓN PROYECTO NODE

Ahora sí, empezamos un **proyecto nuevo** utilizando **npm**:

```
$ npm init
```

Y completamos las cuestiones que se solicitan (se pueden dejar por defecto)

```
package name: (apiRest-producto)
version: (1.0.0)
description: api para clases de SOD
entry point: (index.js)
test command:
git repository: https://iren_lorenzo@bitbucket.org/iren_lorenzo/apiRest-
producto.git
keywords: CRUD, REST, node, express, mongo
author: Iren Lorenzo
license: (ISC)
About to write to
C:\Users\irenf\OneDrive\Documentos\devel\repositorio\api_sod\package.json:
{
  "name": "api_sod",
  "version": "1.0.0",
  "description": "api para clases de SOD",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://iren_lorenzo@bitbucket.org/iren_lorenzo/apiRest-
producto.git"
  },
  "keywords": [
    "CRUD",
    "REST",
    "node",
    "express",
    "mongo"
  ],
  "author": "Iren Lorenzo",
  "license": "ISC",
  "bugs": {
    "url": "https://bitbucket.org/iren_lorenzo/apiRest-producto/issues"
  },
  "homepage": "https://bitbucket.org/iren_lorenzo/apiRest-producto#readme"
}


Is this OK? (yes) y
```

Abrimos el editor de código (**Visual Studio Code**), indicándole que queremos trabajar desde la carpeta actual:

```
$ code .
```

Nota: iniciando nuestro editor con un punto ('.') provoca que se abra en la carpeta actual, lo que facilitará trabajar con múltiples archivos y subcarpetas.

Desde el propio editor podremos comprobar que se ha creado un archivo `package.json` en el directorio de trabajo con la información antes introducida durante la creación del proyecto:

```
{
  "name": "apiRest-producto",
  "version": "1.0.0",
  "description": "api para clases de SOD",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://iren_lorenzo@bitbucket.org/iren_lorenzo/apiRest-
producto.git"
  },
  "keywords": [
    "CRUD",
    "REST",
    "node",
    "express",
    "mongo"
  ],
  "author": "Iren Lorenzo",
  "license": "ISC",
  "bugs": {
    "url": "https://bitbucket.org/iren_lorenzo/apiRest-producto/issues"
  },
  "homepage": "https://bitbucket.org/iren_lorenzo/apiRest-producto#readme"
}
```

Creamos una primera aplicación para comprobar el funcionamiento de **NodeJS** y lo sencillo que es poner en marcha un servidor Web bajo esta tecnología. Para ello creamos dentro de la carpeta del proyecto, con nuestro editor de código, el archivo **index.js** y escribimos en él el siguiente código:

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

const PORT = 8080;
const server = http.createServer();

function HTTP_Response(request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.write('Hola a todas y a todos!\n');
    response.end();
}

server.on('request', HTTP_Response);
server.listen(PORT);

console.log(`Servidor ejecutándose en http://localhost:${PORT}`);
```

Para probarlo, desde la terminal ejecutamos el servidor **NodeJS** con nuestro código `index.js`.

```
$ node index.js
Servidor ejecutándose en http://localhost:8080
```

Ahora podemos conectarnos desde nuestro *navegador web* a través del puerto que hemos establecido (http://localhost:8080/) para ver el resultado.

Nodejs, al basarse en librerías de alto nivel nos abstrae más aún de las llamadas del sistema operativo, en este caso, con la llamada server.listen(PORT); nodejs por debajo crea un socket, hace un bind, un listen y un accept.

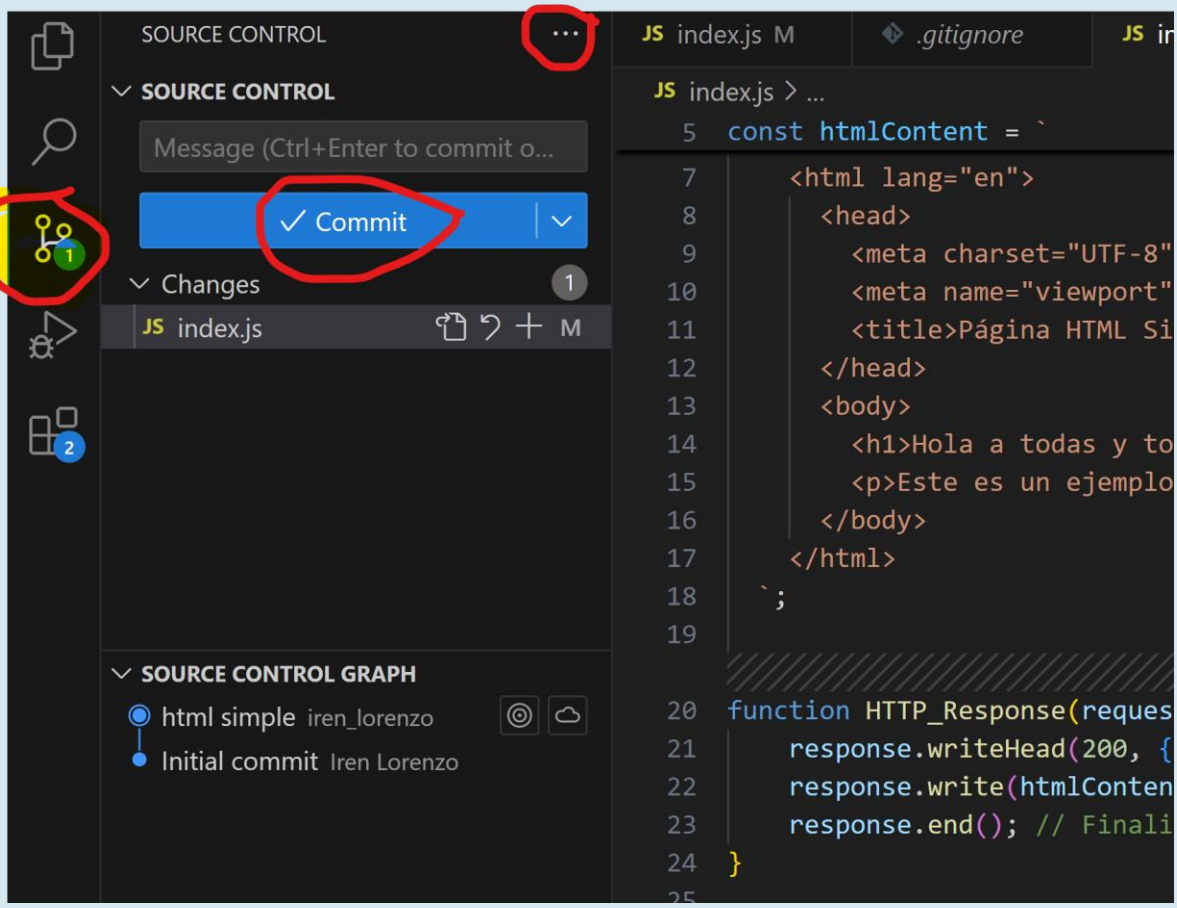
Nota: recordemos ir subiendo una copia al repositorio cada vez que tengamos una versión nueva. Por ejemplo, mediante los comandos:

$ git add .

$ git commit -m "Primer servidor HTTP con NodeJS"

$ git push

También se puede hacer utilizando Visual Studio Code como interfaz gráfica del git para hacer el commit y push



Ahora vamos a hacer una nueva versión más sofisticada del servidor web que hemos hecho donde devuelva una página html. Para ello sustituimos el código actual del index.js por el siguiente:

```
const http = require('http');
const PORT = 8080;

const server = http.createServer();
const htmlContent = `
    <!DOCTYPE html>
    <html lang="en">
      <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Página HTML Simple</title>
      </head>
      <body>
        <h1>Hola a todas y todos!!</h1>
        <p>Este es un ejemplo de una página HTML servida por Node.js.</p>
      </body>
    </html>
  `;

function HTTP_Response(request, response) {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    response.write(htmlContent);
    response.end(); // Finaliza la respuesta
}

server.on('request', HTTP_Response);
server.listen(PORT);

console.log(`Servidor ejecutándose en http://localhost:${PORT}`);
```

Es posible que antes tengamos que detener la versión anterior, si no lo habíamos hecho. Esto se puede hacer presionando la combinación de teclas <Ctrl+C> en la terminal.

Una vez lanzado el nuevo servicio, recargaremos la página desde nuestro navegador Web para verificar que todo funciona correctamente y que se muestra la página html.

Nota: nuevamente, crearemos una copia de la versión en repositorio local y subiremos una copia al repositorio remoto:

```
$ git add .
$ git commit -m "Primer servidor HTTP refactorizado con html"
$ git push
```

O utilizamos Visual Studio Code para ello

En nuestra filosofía de desarrollo vamos a separar el backend del fronted, por tanto, el API no devolverá directamente HTML, sino que devolverá objetos de tipo JSON que serán utilizados luego por el frontend para montar el html. Por eso ahora cambiaremos este código en la siguiente sección.

INSTALACIÓN DE **EXPRESS** PARA **NODEJS**

Express es un **framework web** para **Node.js** que facilita la construcción de aplicaciones web y API RESTful de manera rápida y sencilla. Express proporciona un conjunto de herramientas y funcionalidades que permiten manejar las rutas, solicitudes HTTP, middleware, y demás aspectos fundamentales para desarrollar servidores web. Es uno de los frameworks más populares y utilizados en el entorno de **Node.js** debido a su simplicidad, flexibilidad y gran rendimiento.

A continuación, instalamos y probamos la biblioteca **Express**. Esta biblioteca proporciona una capa adicional sobre **NodeJS** que facilita enormemente la gestión de métodos y recursos HTTP.

```
$ npm i -S express
```

Nota: es la forma simplificada del comando `npm install express --save`.

Esto creará una carpeta **`node_modules`, dentro de la carpeta de nuestro proyecto**.
La opción `-S` (o `--save`) fuerza a que se registre una entrada en el archivo **`package.json`, lo que permitirá posteriormente realizar una fácil instalación de todos los módulos que requiere un proyecto**.

```
"dependencies": {
        "express": "^4.21.1"
}
```

Creamos nuestra primera aplicación **node+express** (si lo deseamos, podemos guardar en el repositorio la versión anterior). Para ello utilizamos el código siguiente en el archivo index.js.

```
'use strict'

const express = require('express');
const app = express();
const PORT = 8080;

app.get('/api/producto', (request, response) => {
        response.send('GET de producto desde Express!');
});

app.listen(8080, () => {
        console.log(`API REST ejecutándose en http://localhost:${PORT}/api/producto`);
});
```

Desde la terminal ejecutamos la aplicación

```
$ node index.js
API REST ejecutándose en http://localhost:8080
^C para cerrar la aplicación
```

Ahora podemos probarlo con nuestro navegador conectándonos a la URL http://localhost:8080/api/producto y el resultado debe ser: GET de producto desde Express!

Nota: nuevamente, crearemos una copia de la versión en repositorio local y subiremos una copia al repositorio remoto:
$ git add .
$ git commit -m "Primer servidor HTTP refactorizado con html"
$ git push

O utilizamos Visual Studio Code para ello

Ampliamos la aplicación anterior modificando **`index.js`** para que soporte el *método HTTP Request* **GET** para obtener un producto concreto, utilizando la ruta `/api/producto/:id`. Es decir, una ruta que comienza por

`/api/producto/`, seguida de cualquier otra cadena que nuestra aplicación interpretará que es un identificador.

```
'use strict'

const express = require('express');
const app = express();

const PORT = 8080;


app.get('/api/producto/:id', (req, res) => {
    res.status(200).send({ mensaje: `Producto con id ${req.params.id}` });
});

app.listen(8080, () => {
      console.log(`API REST ejecutándose en http://localhost:${PORT}/api/producto/:id`);
});
```

Nota 1: al utilizar dos puntos (":") en la especificación de la ruta `/api/producto/:id`, le estamos indicando que "`id`" no es un literal, sino un parámetro variable.

Nota 2: cuando utilicemos *expansión de parámetros* en nuestro código mediante la expresión `${ req.params.id}`, hay que modificar las comillas ' por las comillas ` para que se realice dicha *expansión*.

Nuevamente tendremos que detener desde la terminal el anterior servidor, y volver a lanzar la nueva versión:

```
^C para cerrar la aplicación, si estaba en marcha…
$ node index.js
API REST ejecutándose en http://localhost:8888/api/producto/:id
```

Lo probamos con nuestro navegador conectándonos a la URL: http://localhost:8080/api/producto/45 y el resultado debe ser:

{"mensaje": "Producto con id 45"}

Nota: aunque parezca que el resultado es muy parecido al del ejercicio anterior, nada más lejos de la realidad. En el caso anterior, el mensaje mostrado consistía en una mera cadena de texto. Sin embargo, en este caso, se trata de un verdadero objeto JSON que así reconoce el navegador (en caso de que no lo haga, como se ha indicado anteriormente, es conveniente instalar alguno de los cientos de *plugins* o extensiones que existen para JSON).

Nota: antes de terminar,
Actualizamos la versión editando el archivo `./package.json` a la versión `v1.0.25`.
Esto lo hacemos siguiendo el patrón `MAJOR.MINOR.PATCH`.
A continuación, subimos una copia al repositorio:

```
$ git add .
$ git commit -m "Primer API REST con Express parametrizable"
$ git push
```

O con Visual Studio Code

Ahora, es conveniente crear un tag `en el repositorio con la versión del package.json`. Para ello utilizaremos las etiquetas (**tags**) de la siguiente forma:

```
$ git tag v1.0.25
$ git push -tags
```

Otra opción es crearlo directamente desde el servidor de bitbucket en https://bitbucket.org/. Ir al commit desde el que queremos crear el tag, hacer clic y en el menú de la derecha hacer clic en Create tag.

INSTALACIÓN Y EJECUCIÓN DE MONGODB 

Con el objetivo de hacer persistente en Base de Datos la información del API, debemos instalar un gestor de base de datos. En este caso, como es parte de la pila MEAN, instalaremos MongoDB.

MongoDB es una base de datos NoSQL orientada a documentos que permite almacenar y consultar datos en un formato flexible y escalable. A diferencia de las bases de datos relacionales tradicionales que utilizan tablas y filas, MongoDB almacena datos en documentos BSON (una representación binaria de JSON - JavaScript Object Notation).

Paso 1: Descargar MongoDB

1. **Ir al sitio de descargas de MongoDB:**
   - Abre tu navegador web y visita la página oficial de descargas de MongoDB: MongoDB Download Center.
2. **Seleccionar la versión para Windows:**
   - En la sección de "Community Server", selecciona la versión de MongoDB adecuada para tu sistema operativo (Windows).
   - Asegúrate de seleccionar el instalador MSI, que es el más fácil de usar en Windows.
3. **Descargar el instalador:**
   - Haz clic en el botón "Download" para descargar el instalador MSI.

Paso 2: Instalar MongoDB

1. **Ejecutar el instalador:**
   - Una vez que la descarga esté completa, ejecuta el archivo MSI descargado.
2. **Configurar el servicio MongoDB:**
   - En la sección "Service Configuration", asegúrate de que las opciones "Install MongoDB as a Service" (significa que cada vez que se inicie Windows se levantará el servidor de mongodb) y "Run service as Network Service user" estén seleccionadas.
   - Puedes dejar el nombre del servicio como está, que generalmente es "MongoDB".
3. **Finalizar la instalación:**
   - Haz clic en "Next" y luego en "Install" para comenzar la instalación.
   - Una vez que la instalación esté completa, haz clic en "Finish".
4. Además del servidor se instala un cliente de mongo (Compass) con el cual podremos ver los cambios en la Base de datos

Paso 3: Verificar el Servicio de Windows

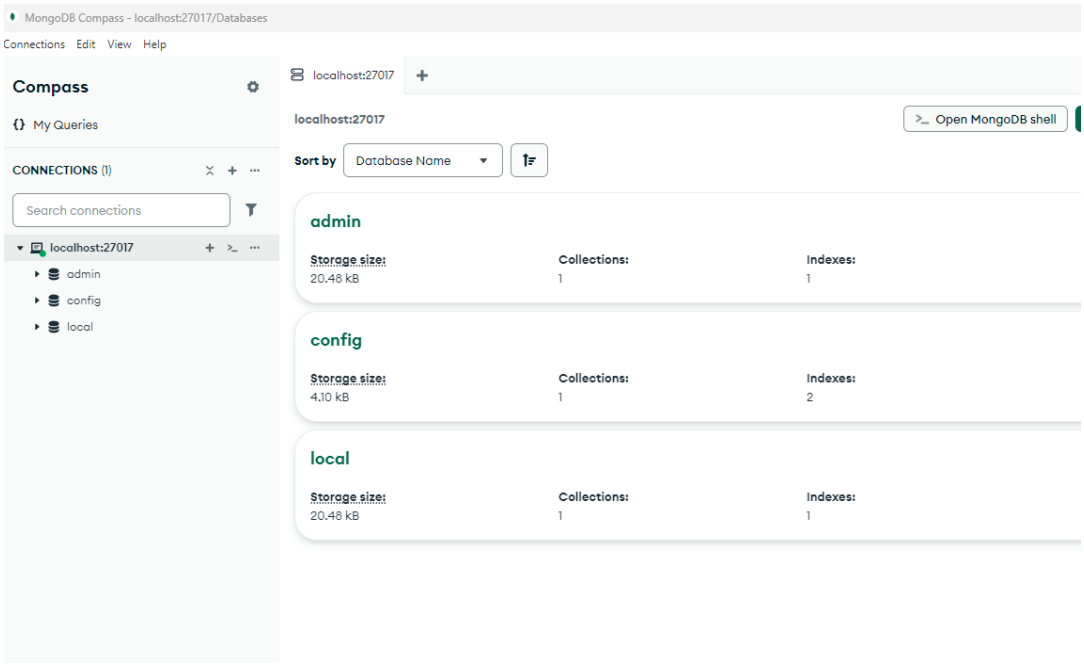
Si has instalado MongoDB como un servicio en Windows, puedes verificar su estado utilizando la consola de servicios de Windows.

1. **Abrir la consola de servicios**:
   - Presiona Win + R, escribe services.msc y presiona Enter.
2. **Buscar el servicio MongoDB**:
   - En la lista de servicios, busca un servicio llamado "MongoDB" o "MongoDB Server".
3. **Verificar el estado del servicio**:

- Asegúrate de que el estado del servicio es "En ejecución". Si está detenido, puedes iniciarlo haciendo clic derecho sobre él y seleccionando "Iniciar".

Paso 4: Verificar con Compass

1. Abrir compass como cliente de BD de mongo
2. Conectarse al servidor que está en localhost:27017 (conexión por defecto)
3. Ver las bases de datos por defecto que hay en el servidor:



IMPLEMENTACIÓN DEL SERVICIO CRUD

Dentro del proyecto tendremos que instalar la biblioteca **mongodb** para trabajar desde nuestro proyecto con esta base de datos y, en nuestro caso, la biblioteca **mongoose** que nos simplificará la forma de acceder a **MongoDB** desde nuestro código **javascript**.
Para ello estando en consola dentro de la carpeta del proyecto del proyecto, debemos usar el gestor de librerías npm para la instalación de ambas:

```
$ npm i -S mongodb
$ npm i -S mongoose
```

Verificamos la instalación en el package.json:

```
"dependencies": {
  "express": "^4.21.1",
  "mongodb": "^6.11.0",
  "mongoose": "^8.8.2"
 }
```

Una vez tenemos todas las herramientas y bibliotecas que necesitamos, reescribiremos en el archivo **`index.js`** el código **JavaScript** que implementará la funcionalidad esperada de nuestro servidor.

Primero, necesitamos establecer una conexión con el servidor MongoDB agregando la configuración de mongoose en index.js:

```
...
```

```
const PORT = 8080;

const mongoose = require('mongoose');


// Conectar a MongoDB
mongoose.connect('mongodb://localhost:27017/producto')
.then(() => {
        console.log('Conexión a la base de datos establecida');
}).catch(err => {
        console.error('Error de conexión a la base de datos:', err);
});

…
```

Nota: en ese caso hemos llamado a la DB producto (como se ve en la URL de conexión), pero puede ser cualquier nombre que selecciones.

Si desde consola ejecutamos el API:

node index.js

Veremos que obtenemos el mensaje de conexión a la BD:

```
C:\Users\irenf\Documents\devel\desarrollo\repositorio\apiSod\api_sod>node index.js
API REST ejecutándose en http://localhost:8080/api/producto/:id
Conexión a la base de datos establecida
```

Nota: Este es un buen momento para hacer un commit al git con el texto: "Conexión a Base de datos Mongo".
Se puede hacer desde líneas de comando del git o desde visual studio code.

Una vez implementada la conexión a la BD, como estamos utilizando mogoose, debemos crear un modelo de que recoja la información que se almacenará en la base de datos. Para ello, con el objetivo de organizar el código, crearemos una carpeta llamada models y dentro el archivo producto.js con el código siguiente:

```
'use strict'

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const ProductoSchema = new Schema({
    nombre: { type: String, required: true },
    precio: { type: Number, required: true },
    descripcion: String,
    categoria: String,
    stock: Number
});

module.exports = mongoose.model('Producto', ProductoSchema);
```

En esta líneas se incluye la librería de mogoose, así como la de schemas. Luego se crea el esquema con los campos del producto: nombre, precio, descripción, categoría y stock. Por último, se exporta el esquema bajo en nombre Producto, para que pueda ser utilizado en otros archivos.
Para que pueda ser utilizado el esquema en index.js, debemos incorporar esta línea en la zona de constantes:

```
const Producto = require('./models/producto'); // Importar el modelo
```

En este momento el index.js debe quedar de esta manera:

```
'use strict'

const express = require('express');
const app = express();
const mongoose = require('mongoose');
const Producto = require('./models/producto'); // Importar el modelo
const PORT = 8080;

// Conectar a MongoDB
mongoose.connect('mongodb://localhost:27017/producto')
.then(() => {
    console.log('Conexión a la base de datos establecida');
}).catch(err => {
    console.error('Error de conexión a la base de datos:', err);
});



app.get('/api/producto/:id', (req, res) => {
    res.status(200).send({ mensaje: `Producto con id ${req.params.id}` });
});

app.listen(8080, () => {
    console.log(`API REST ejecutándose en http://localhost:${PORT}/api/producto/:id`);
});
```

Comprobamos nuevamente que todo funciona con: node index.js

```
C:\Users\irenf\Documents\devel\desarrollo\repositorio\apiSod\api_sod>node index.js
API REST ejecutándose en http://localhost:8080/api/producto/:id
Conexión a la base de datos establecida
```

Para la creación de nuestro API CRUD de productos, vamos a implementar 5 endpoints principales:

1. GET /api/producto/:id -> con el que obtendremos los datos del producto con el identificador solicitado
2. GET /api/producto -> con el que obtendremos un listado con todos los productos
3. POST /api/producto -> con el que crearemos un nuevo producto. Los datos del nuevo producto llegarán al API en el body del HTTP Request
4. PUT /api/producto/:id -> con el que actualizaremos los datos del producto con el identificador solicitado. Los datos para actualizar del producto llegarán al API en el body del HTTP Request
5. DELETE /api/producto/:id -> con el que eliminaremos el producto con el identificador solicitado

Para que en las llamadas POST y PUT sea fácil interpretar los datos que llegan por el body de la request en formato json, debemos añadir al index.js, después del código de conexión a mongo, la siguiente línea:

```
app.use(express.json());
```

El middleware app.use(express.json()) se utiliza para habilitar el análisis (parseo) del cuerpo de las solicitudes HTTP con formato JSON. Esto es necesario cuando recibes datos en formato JSON desde el cliente (como en una solicitud POST o PUT) y quieres que esos datos estén disponibles en el cuerpo de la solicitud (req.body) en tu aplicación Express.

Para implementar el primer endpoint, debemos sustituir el fragmento de código existente en el index.js:

```
app.get('/api/producto/:id', (req, res) => {
    res.status(200).send({ mensaje: `Producto con id ${req.params.id}` });
```

```
});
```

Por el código necesario para que moongose busque en la base de datos el producto utilizando el identificador que llega a traves del parámetro de la request:

```
// Ruta para obtener un producto por id
app.get('/api/producto/:id', async (req, res) => {
    let productoId = req.params.id;

    try {
        const producto = await Producto.findById(productoId);

        if (!producto) {
            return res.status(404).send({ mensaje: 'El producto no existe' });
        }

        res.status(200).send({ producto });
    } catch (err) {
        res.status(500).send({ mensaje: `Error al realizar la petición: ${err}` });
    }
});
```

Nota: el acceso de nodejs a la base de datos es asíncrona. Node.js no utiliza múltiples hilos directamente para ejecutar código JavaScript, pero internamente hace uso de hilos a través de libuv, una biblioteca de C que maneja operaciones de I/O asincrónicas en segundo plano.

- Operaciones de I/O: Operaciones como la lectura de archivos, la conexión a bases de datos, y las solicitudes HTTP son delegadas a hilos del sistema operativo a través de libuv.

Al ser una llamada asíncrona se ejecuta con el await, esto hace que la ejecución espere a que termine la llamada a base de datos para continuar con la ejecución del programa.
La función **findById**, devuelve lo que se conoce en javascript como un objeto promesa. Una promesa (promise) es un objeto que representa la eventual finalización (o falla) de una operación asincrónica y su valor resultante.
Para manejar el resultado de una promesa, se utilizan los métodos then, catch y finally.

- then: Se ejecuta cuando la promesa se cumple exitosamente.
- catch: Se ejecuta cuando la promesa es rechazada.
- finally: Se ejecuta independientemente de si la promesa fue cumplida o rechazada.

El siguiente código es análogo al anterior, pero utilizando el objeto promesa y sus métodos:

```
// Ruta para obtener un producto por id
app.get('/api/producto/:id', (req, res) => {
  let productoId = req.params.id;

  Producto.findById(productoId)
    .then(producto => {
      if (!producto) {
        return res.status(404).send({ mensaje: 'El producto no existe' });
      }
      res.status(200).send({ producto });
    })
    .catch(err => {
      res.status(500).send({ mensaje: `Error al realizar la petición: ${err}` });
    });
});
```

Si volvemos a ejecutar el servidor y hacemos la misma llamada en el navegador http://localhost:8080/api/producto/45 obtendremos el sigueinte mensaje:

```
{
"mensaje": "Error al realizar la petición: CastError: Cast to ObjectId failed for value
\"45\" (type string) at path \"_id\" for model \"Producto\""
}
```

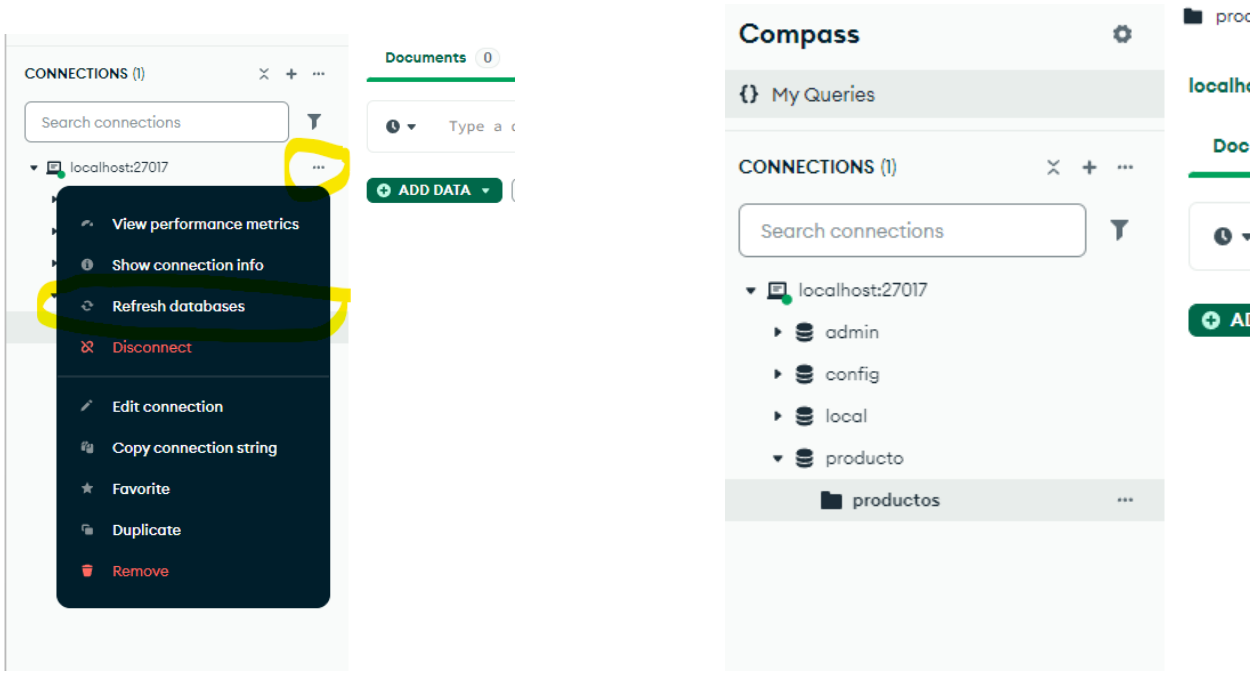
Si observamos el modelo de producto no hemos incluido ningún id dentro de los campos, pero mongoos de encarga de crear por defecto el campo _id que es de tipo ObjectId y como 45 no cumple con el tipo esperado, obtenemos un error de tipo. Sin embargo, si probamos la siguiente llamada:

http://localhost:8080/api/producto/60c72b2f9b1e8cfa3a7e7c88

Obtendremos el error que hemos programado para cuando no exista producto con ese identificador:

```
{
	"mensaje": "El producto no existe"
}
```

En este momento se vemos la base de datos utilizando Compass, veremos que el API a través de la librería de mongoose ha creado la base de datos "producto". Si teníamos el compass abierto habría que refrescar para verla:



Nota: Este es un buen momento para hacer un commit al git con el texto: "Endpoint get producto por id".
Se puede hacer desde líneas de comando del git o desde visual studio code.

El siguiente endpoint que haremos será el de crear un producto en la base de datos, para ello debemos añadir el siguiente código en index.js:

```
// Ruta para crear un nuevo producto
app.post('/api/producto', async (req, res) => {
    console.log('POST /api/producto');
    console.log(req.body);

    // Crear un nuevo producto utilizando req.body
    let producto = new Producto(req.body);
```

```
    try {
        // Guardar el producto en la base de datos
        const productoStored = await producto.save();
        res.status(200).send({ producto: productoStored });
    } catch (err) {
        res.status(500).send({ mensaje: `Error al salvar en la base de datos: ${err}` });
    }
});
```

Al igual que en el ejemplo anterior se comienza con una definición de la ruta:

```
app.post('/api/producto', async (req, res) => {
```

- app.post: Define una ruta que manejará las solicitudes HTTP POST. POST se utiliza típicamente para crear nuevos recursos en el servidor.
- '/api/producto': Especifica el endpoint donde esta ruta estará disponible.
- async (req, res): Define una función asincrónica que maneja la solicitud (req) y la respuesta (res).

La dos líneas siguientes son para hacer log desde el servidor de lo que se va realizando:

```
console.log('POST /api/producto');
console.log(req.body);
```

- console.log('POST /api/producto'): Imprime en la consola que se ha recibido una solicitud POST en /api/producto.
- console.log(req.body): Imprime el cuerpo de la solicitud en la consola. Esto es útil para depuración y verificación de los datos que se están enviando al servidor.

En la siguiente línea se crea un objeto de tipo Producto:

```
let producto = new Producto(req.body);
```

- let producto = new Producto(req.body): Crea una nueva instancia del modelo Producto utilizando los datos enviados en el cuerpo de la solicitud (req.body). Se asume que req.body tiene la estructura correcta para crear un Producto según el esquema definido en Mongoose. Para evitar errores so podría asignar valor a valor.

Las siguientes líneas guardan el producto en la BD:

```
 try {
        // Guardar el producto en la base de datos
        const productoStored = await producto.save();
        res.status(200).send({ producto: productoStored });
    } catch (err) {
        res.status(500).send({ mensaje: `Error al salvar en la base de datos: ${err}` });
    }
```

try / catch: Se utiliza para manejar errores de manera asíncrona.

- **await producto.save()**: Intenta guardar el nuevo producto en la base de datos. La palabra clave await se utiliza para esperar a que la promesa se resuelva. Si la operación es exitosa, se almacena en productoStored.
- **res.status(200).send({ producto: productoStored })**: Si el guardado es exitoso, se envía una respuesta al cliente con el estado HTTP 200 (OK) y el producto guardado.
- **catch (err)**: Si ocurre un error durante el guardado del producto, se captura y maneja aquí.
    - **res.status(500).send({ mensaje: Error al salvar en la base de datos: ${err} })**: Si hay un error, se envía una respuesta al cliente con el estado HTTP 500 (Internal Server Error) y un mensaje de error.

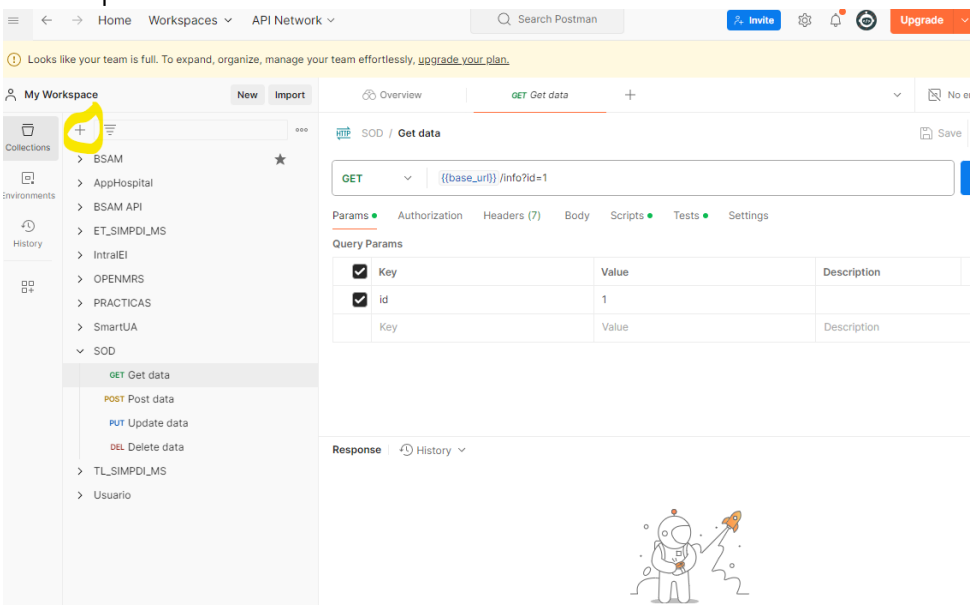
Nota: Este es un buen momento para hacer un commit al git con el texto: "Endpoint post para crear producto".
Se puede hacer desde líneas de comando del git o desde visual studio code.

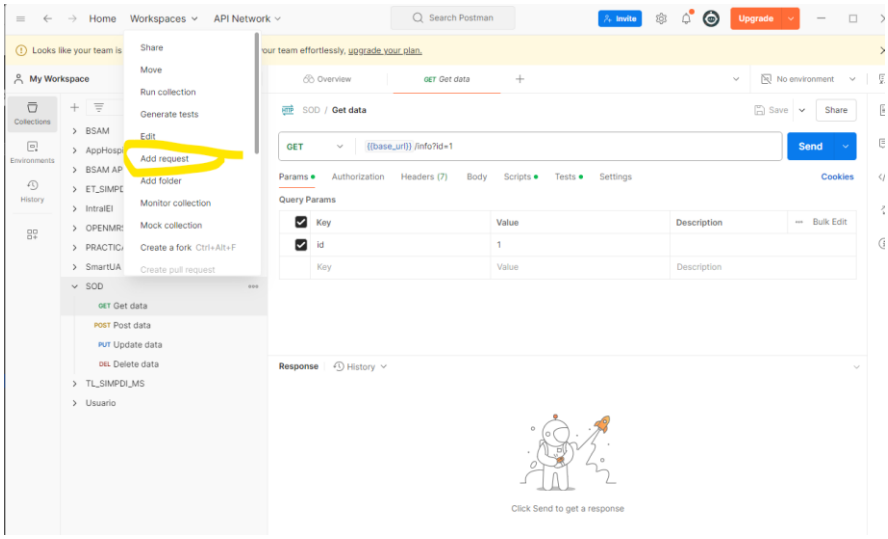
PROBANDO CON POSTMAN

Hasta ahora hemos probado la salida del API utilizando el navegador. Esto ha sido posible porque hasta ahora todas eran tipo GET.
Cuando navegas a una URL en un navegador (como Chrome, Firefox, etc.), el navegador realiza una solicitud HTTP GET. Esta es la solicitud predeterminada cuando ingresas una dirección web en la barra de direcciones.
Para enviar datos al servidor utilizando el método POST, necesitas enviar un cuerpo de solicitud. Los navegadores no tienen una interfaz nativa para realizar solicitudes POST desde la barra de direcciones; están diseñados principalmente para GET.
Postman es una herramienta que te permite crear y enviar fácilmente solicitudes HTTP de diferentes tipos (GET, POST, PUT, DELETE, etc.) con un cuerpo de solicitud personalizado, encabezados, autenticación, y más.

Instalar Postman

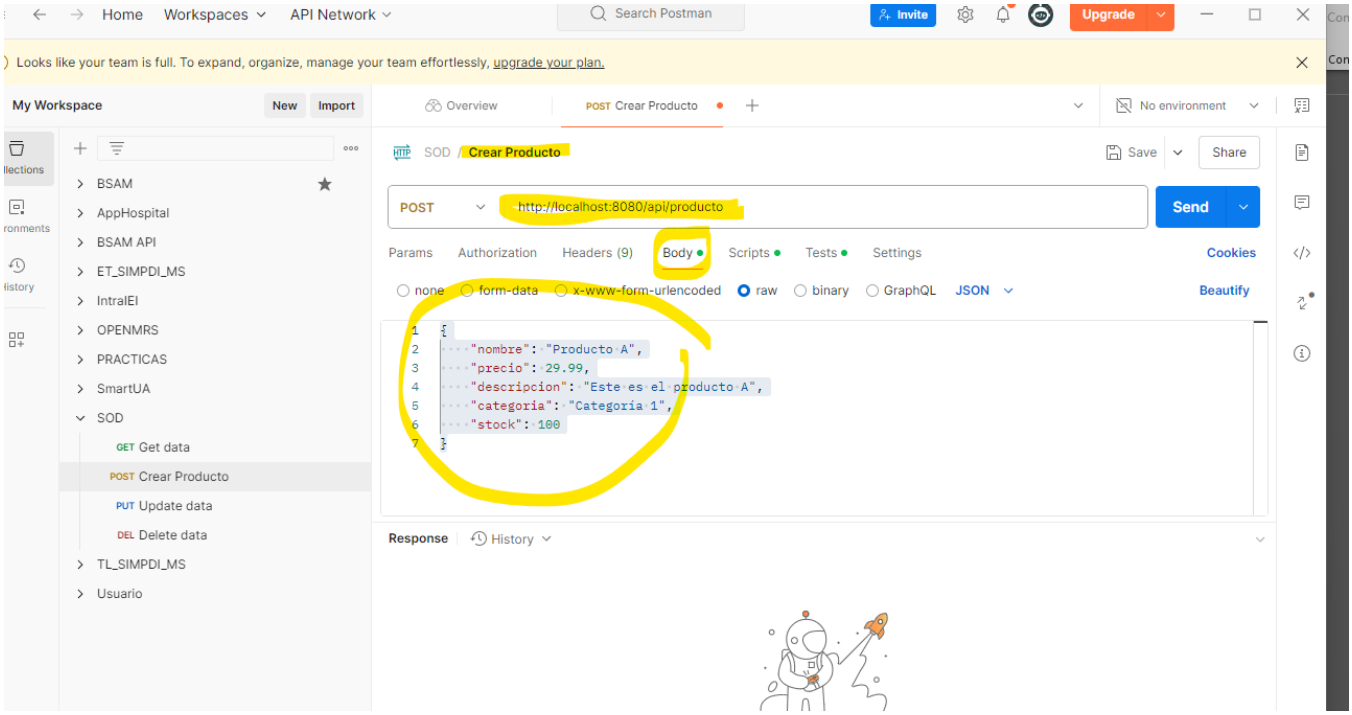
Descarga el Postman desde: https://www.postman.com/downloads/ , has doble clic y sigue las instrucciones de instalación.
Cuando abras Postman por primera vez, se te pedirá que inicies sesión o crees una cuenta. Puedes crear una cuenta gratuita si no tienes una. También puedes optar por usar Postman sin una cuenta seleccionando la opción de continuar sin una cuenta.
Con el objetivo de organizar el trabajo, crea una nueva colección en Postman llamada SOD y dentro puedes ir creando las distintas request.



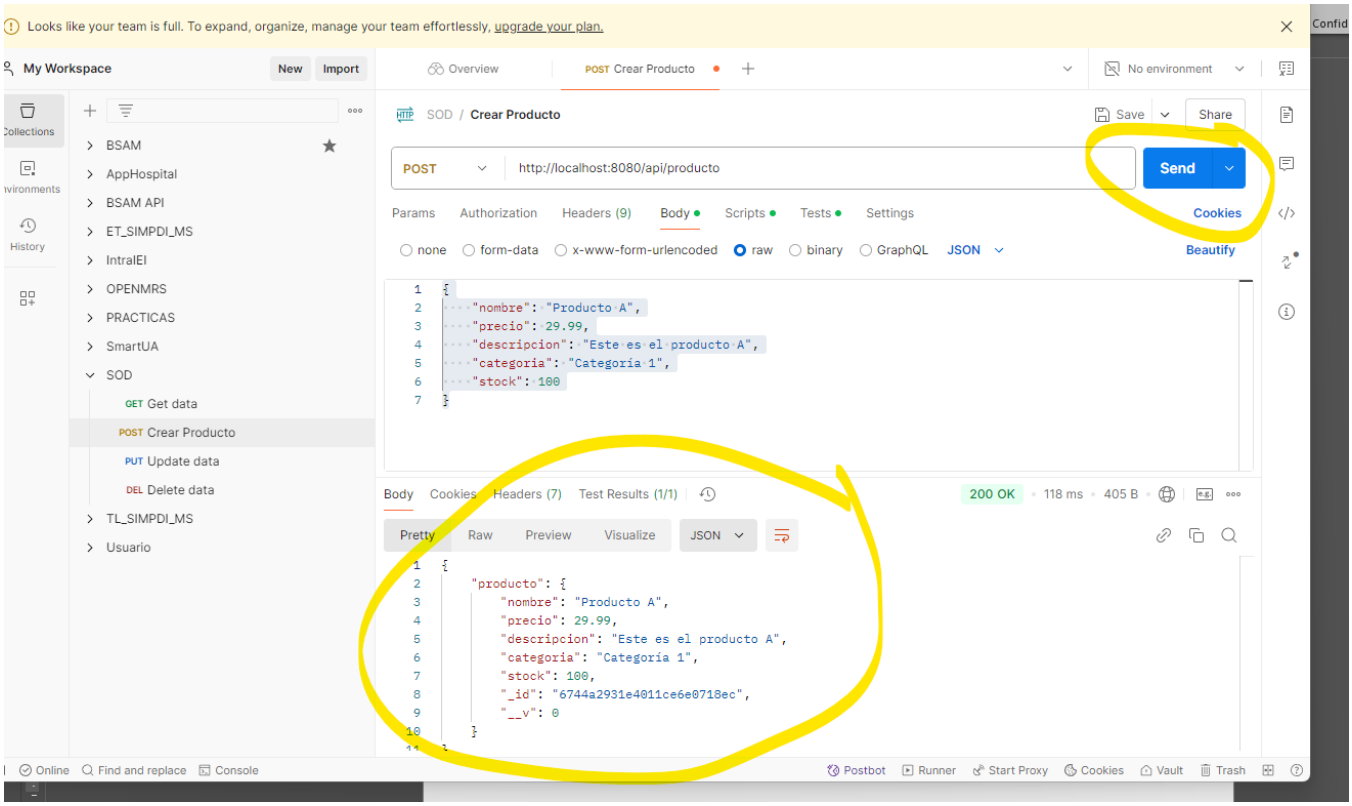


Empezamos por el POST para poder crear nuevos productos.
En el body de la request podemos pasar el siguiente json con datos de ejemplo par aun producto:

```
{
    "nombre": "Producto A",
    "precio": 29.99,
    "descripcion": "Este es el producto A",
    "categoria": "Categoría 1",
    "stock": 100
}
```



Teniendo el API en ejecución (node index.js) si hacemos clic en send, veremos en resultado la zona que muestra el body del response:



Podemos verificar la base de datos utilizando el Comapss y veremos que el producto se ha creado correctamente. Si ya teníamos el Compass abierto es necesario refrescar.

REFACTORIZACIÓN

Hemos desarrollado ambos endpoints en el index.js con objetivos docentes ya que es más sencillo ir entendiendo las fases. No obstante, antes de continuar con los siguientes, vamos a refactorizar el código para que quede con la estructura profesional que debe tener.

Aunque el API lo desarrollaremos en un único componente, es importante tener una estructura correcta para que sea fácil de mantener. Para ello, la implementación la dividiremos en capas:

- Controladores: serán las funciones encargadas de gestionar las solicitudes HTTP y ofrecer las respuestas a los clientes que soliciten información del API.
- Servicios: serán las funciones encargadas de la lógica del negocio y el acceso a base de datos
- Modelos: son los modelos de base de datos.

La estructura del proyecto debe quedar así:

```
/controllers
    productoController.js
/models
    producto.js
/services
    productoService.js
index.js
```

index.js y la carpeta models, ya la teníamos creadas por tanto habría que crear las restantes dos carpetas con sus ficheros js en blanco dentro.

Ahora debemos actualizar el fichero de productoServices.js, son todo el código necesario para el acceso a base de datos:

```
// services/productoService.js
'use strict';

const Producto = require('../models/producto');

async function obtenerProductoPorId(id) {
    try {
        return await Producto.findById(id);
    } catch (err) {
        throw new Error(`Error al obtener el producto: ${err}`);
    }
}

async function crearProducto(datosProducto) {
    try {
        const producto = new Producto(datosProducto);
        return await producto.save();
    } catch (err) {
        throw new Error(`Error al crear el producto: ${err}`);
    }
}


module.exports = {
    obtenerProductoPorId,
    crearProducto
};
```

Ahora en el fichero productoController.js, debemos poner toda la parte del código que teníamos en el index que gestionaba las solicitudes http, y llamar al servicio para obtener las respuesta:

```
// controllers/productoController.js
'use strict';

const productoService = require('../services/productoService');

async function obtenerProductoPorId (req, res)  {
    let productoId = req.params.id;

    try {
        const producto = await productoService.obtenerProductoPorId(productoId);
        if (!producto) {
            return res.status(404).send({ mensaje: 'El producto no existe' });
        }
        res.status(200).send({ producto });
    } catch (err) {
        res.status(500).send({ mensaje: `Error al realizar la petición: ${err.message}` });
    }
};


async function crearProducto (req, res) {
    console.log('POST /api/producto');
    console.log(req.body);

    try {
        const productoStored = await productoService.crearProducto(req.body);
        res.status(200).send({ producto: productoStored });
    } catch (err) {
        res.status(500).send({ mensaje: `Error al salvar en la base de datos: ${err.message}`
});
    }
};
```

```
module.exports = {
    obtenerProductoPorId,
    crearProducto
};
```

Nota: Las funciones de ambos archivos están creadas de la forma tradicional porque es más parecido a lo que hemos estado trabajando hasta ahora. No obstante, javascript tiene una sintaxis más compacta para declarar funciones, llamada función flecha.

Las funciones tradicionales se declaran utilizando la palabra clave function. Por ejemplo:

```
function obtenerProductoPorId(id) {
  // cuerpo de la función
}
```

Las funciones flecha tienen una sintaxis más corta y se escriben así:

```
const obtenerProductoPorId = (id) => {
  // cuerpo de la función
};
```

Entonces la función obtenerProductoPorId, del servicio puede escribirse de esta forma:

```
async function obtenerProductoPorId(id) {
  try {
    return await Producto.findById(id);
  } catch (err) {
    throw new Error(`Error al obtener el producto: ${err}`);
  }
}
```

O utilizando la función flecha de la siguiente forma:

```
const obtenerProductoPorId = async (id) => {
  try {
    return await Producto.findById(id);
  } catch (err) {
    throw new Error(`Error al obtener el producto: ${err}`);
  }
};
```

Tenedlo en cuenta porque cuando preguntéis a chatGPT puede devolveros esta sintaxis y quiero que entendáis que es lo mismo :-)

Por último, habría que rescribir el index.js para eliminar el código de gestión de las solicitudes http y de acceso a base de datos. Quedaría como sigue:

// index.js

```
'use strict';

const express = require('express');
const mongoose = require('mongoose');
const productoController = require('./controllers/productoController'); // Importar el
controlador
const app = express();
const PORT = 8080;

// Conectar a MongoDB
mongoose.connect('mongodb://localhost:27017/producto')
.then(() => {
    console.log('Conexión a la base de datos establecida');
}).catch(err => {
    console.error('Error de conexión a la base de datos:', err);
});

app.use(express.json());

// Definir las rutas y asignarles los métodos del controlador
app.get('/api/producto/:id', productoController.obtenerProductoPorId);
app.post('/api/producto', productoController.crearProducto);

app.listen(PORT, () => {
    console.log(`API REST ejecutándose en http://localhost:${PORT}`);
});
```

Nota: Este es un buen momento para hacer un commit al git con el texto: "Codigo Refactorizado".
Se puede hacer desde líneas de comando del git o desde visual studio code.

Ahora con el código ya refactorizado vamos a añadir el endpoint de actualizar:

- PUT /api/producto/:id -> con el que actualizaremos los datos del producto con el identificador solicitado. Los datos para actualizar del producto llegarán al API en el body del HTTP Request

Para ello vamos al index.js y añadimos una nueva ruta:

```
app.put('/api/producto/:id', productoController.actualizarProducto);
```

Ahora tendremos que implementar la función actualizarProducto en el controlador. Así que añadimos esta función en controllers/productoController.js:

```
async function actualizarProducto(req, res) {
    console.log('PUT /api/producto/:id');
    console.log(req.body);

    let productoId = req.params.id;
    let update = req.body;

    try {
        const productoUpdated = await productoService.actualizarProducto(productoId, update);
        if (!productoUpdated) {
            return res.status(404).send({ mensaje: 'El producto no existe' });
        }
        res.status(200).send({ producto: productoUpdated });
    } catch (err) {
        res.status(500).send({ mensaje: `Error al actualizar el producto: ${err.message}` });
    }
};
```

Además de implementar la función, hay que exportarla como módulo al final del fichero, donde debe quedar de esta forma:

```
module.exports = {
    obtenerProductoPorId,
    crearProducto,
    actualizarProducto
};
```

Como vemos desde el controlador se llama a la función actualizarProducto del servicio, por lo que debemos implementarla en: services/productoService.js dicha función:

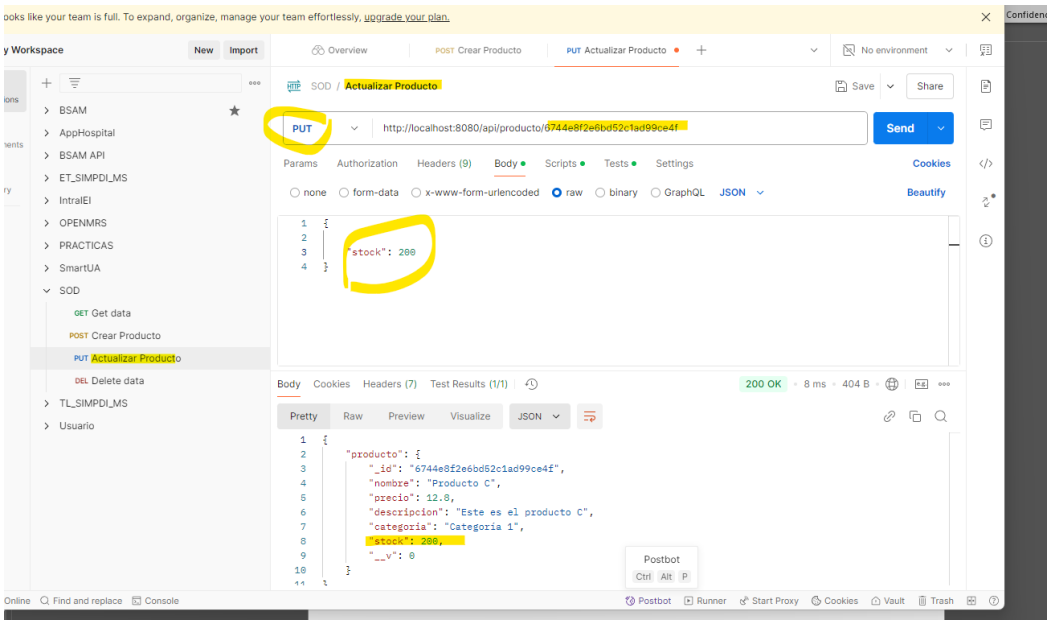
```
async function actualizarProducto(id, datosProducto) {
    try {
        return await Producto.findByIdAndUpdate(id, datosProducto, { new: true });
    } catch (err) {
        throw new Error(`Error al actualizar el producto: ${err}`);
    }
};
```

En este caso la función que nos ofrece mongoose sobre el esquema es findByIdAndUpdate, que permite pasar el identificador del producto y actualizarlos con los valores que lleguen en datosProducto, que como vemos en el controlador, son los valores que llegan por al body a la request. Por último la sintaxis: { new: true }, es para que la función de mongoose devuelva el objeto nuevo ya actualizado, no el antiguo.

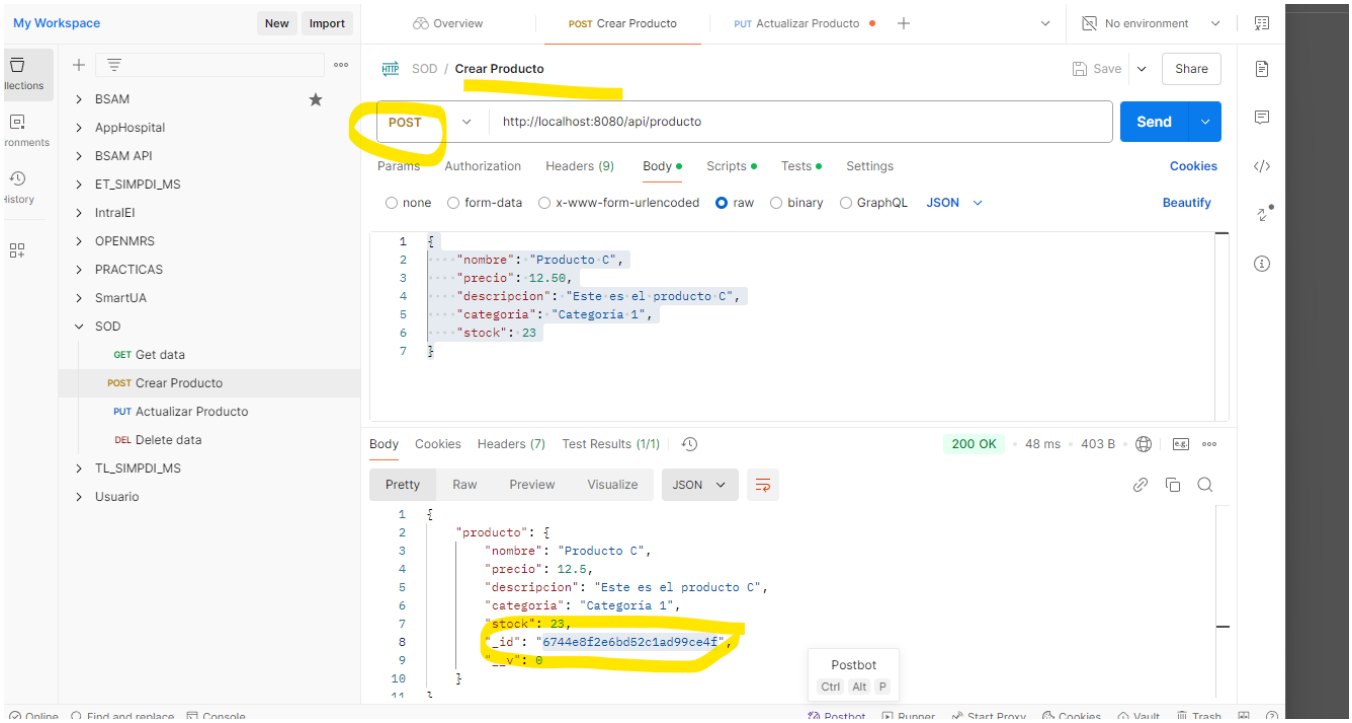
Igualmente hay que exportar la función en los módulos la final del archivo:

```
module.exports = {
    obtenerProductoPorId,
    crearProducto,
    actualizarProducto
};
```

Podemos probar el update desde Postman, creando una request de tipo PUT:



El identificador lo podemos sacar del resultado de un POST, de creación:



Luego en el body, en formato json podemos pasar los campos que queramos modificar.

Sabiendo que desde mongoose tenemos la función find() que devuelve todos los elementos de una colección y la función findByIdAndDelete(id) que busca el elemento con la id pasad por parámetro y lo elimina, implemente dentro del código refactorizado los dos endpoints que faltan:

- GET /api/producto -> con el que obtendremos un listado con todos los productos
- DELETE /api/producto/:id -> con el que eliminaremos el producto con el identificador solicitado

Debe probarlo mediante requests de Postman.

INTERFAZ GRÁFICA
Una vez que tenemos el backend terminado debemos hacer la aplicación de frontend.
Siguiendo con el uso de la pila MEAN, esta deberíamos hacerla con Angular que es el framework para generar aplicaciones del lado del cliente optimizadas con un servidor nodejs. No obstante, por cuestiones de tiempo, y también con el objetivo de que seamos conscientes de la versatilidad de separar backend y frontend, haremos una interfaz sencilla utilizando html y js básico.

Primero creamos una carpeta que se llame frontend (fuera del proyecto del API), abrimos Visual Studio (. code) y creamos un fichero llamado index.html dentro de la carpeta.
En este archivo podemos poner el siguiente código html con:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Lista de Productos</title>
</head>
<body>

    <h1>Productos</h1>
    <table border="1">
```

```
        <thead>
            <tr>
                <th>ID</th>
                <th>Nombre</th>
                <th>Descripción</th>
                <th>Precio</th>
                <th>Categoría</th>
            </tr>
        </thead>
        <tbody id="productos">
            <!-- Los productos se insertarán aquí -->
        </tbody>
    </table>

</body>
</html>
```

Como vemos es sencillamente la estructura html con los encabezados de una tabla:

- Se define una estructura HTML básica con una tabla que contiene una cabecera (thead) y un cuerpo (tbody).
- El cuerpo de la tabla tiene el atributo id="productos" donde se insertarán los productos obtenidos del API.

Si abrimos el html en un navegador veríamos la estructura de la tabla con sus cabeceras:

Productos

ID	Nombre	Descripción	Precio	Categoría

Este html puede mejorar su aspectos utilizando estilos con el CSS. Para hacerlo de forma más profesional, vamos a utilizar Bootstrap.
Bootstrap es un framework de código abierto para el desarrollo de interfaces de usuario (UI) y sitios web responsivos. Fue creado por Mark Otto y Jacob Thornton en Twitter y se lanzó inicialmente en 2011. Bootstrap es muy popular entre desarrolladores web debido a su facilidad de uso y la cantidad de herramientas que ofrece para crear diseños web modernos y consistentes .
Este framework cuenta con sus CSS y javascripts que podemos incluir en nuestro html de forma muy sencilla para obtener resultados más profesiones. En esta práctica haremos un uso muy básico de Bootstrap, pero toda la información de su uso se puede encontrar aquí: https://getbootstrap.com/docs/5.3/getting-started/introduction/
Haciendo estos cambios en el código: añadir el css de Bootstrap, sus 3 scripts básicos y añadir las clases que se encuentran el en CSS a los distintas etiquetas html podemos obtener un resultado mucho más vistoso:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Lista de Productos</title>
    <!-- Enlace a Bootstrap CSS -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>
```

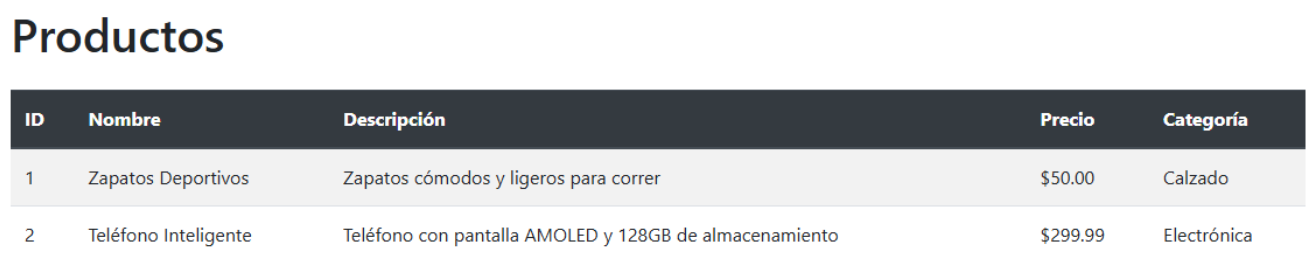
```
    <div class="container mt-5">
        <h1 class="mb-4">Productos</h1>
        <table class="table table-striped">
            <thead class="thead-dark">
                <tr>
                    <th>ID</th>
                    <th>Nombre</th>
                    <th>Descripción</th>
                    <th>Precio</th>
                    <th>Categoría</th>
                </tr>
            </thead>
            <tbody id="productos">
                <!-- Los productos se insertarán aquí -->
            </tbody>
        </table>
    </div>

    <!-- Enlace a jQuery y Bootstrap JS -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.4/dist/umd/popper.min.js"></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

</body>
</html>
```

Este debe ser el resultado:

Fijaos que, al haber incluido a las clases de la tabla, la clase de Bootstrap table-striped, cuando añadamos elementos a la tabla van a ir alternándose en tonalidad tal como tiene definido Bootstrap. Por ejemplo, si añadimos de forma estática elementos a la tabla de esta forma:

```
   <tbody id="productos">

                <!-- Los productos se insertarán aquí -->

                <tr>
                    <td>1</td>
                    <td>Zapatos Deportivos</td>
                    <td>Zapatos cómodos y ligeros para correr</td>
                    <td>$50.00</td>
                    <td>Calzado</td>
                </tr>
                <!-- Segunda fila inventada -->
                <tr>
                    <td>2</td>
                    <td>Teléfono Inteligente</td>
                    <td>Teléfono con pantalla AMOLED y 128GB de almacenamiento</td>
                    <td>$299.99</td>
                    <td>Electrónica</td>
                </tr>
            </tbody>
```

Recargando la página en el navegador tendríamos este resultado:



Productos

ID	Nombre	Descripción	Precio	Categoría
1	Zapatos Deportivos	Zapatos cómodos y ligeros para correr	$50.00	Calzado
2	Teléfono Inteligente	Teléfono con pantalla AMOLED y 128GB de almacenamiento	$299.99	Electrónica

Por otra parte, si queremos más dinamismo, podemos añadir a la etiqueta tabla la clase table-hover. De esta forma se resaltará la fila en la que esté el ratón:

```
<table class="table table-striped table-hover">
```

Para explorar las diferentes funcionalidades de Bootstrap os recomiendo ver su documentación (: https://getbootstrap.com/docs/5.3/getting-started/introduction/
)

AÑADIR DINAMISMO
Ya tenemos la parte estática del frontend, ahora faltaría rellenar los elementos de la tabla de forma dinámica. Es decir, llamar a nuestra API y sacar de su respuesta los valores que rellenarán la tabla.
La parte dinámica de las aplicaciones que se ejecutan en los navegadores (aplicaciones cliente) se implementa utilizando javascript.
Javascript, utilizando el objeto DOM, puede acceder a los diferentes elementos del html y variar su contenido.
Por otra parte, javascript nos permite hacer llamadas asíncronas a servicios a través de la función fetch.
La función fetch en JavaScript se utiliza para realizar solicitudes HTTP a servidores web. Es parte de la API Fetch, una API moderna basada en promesas que permite a los desarrolladores hacer peticiones de red de manera sencilla y manejable.
A continuación, se incluye el código html en el que hemos trabajado (eliminando las filas de prueba de la tabla) y añadiendo el script que utilizando el objecto document (DOM) y fetch rellenan los datos de manera dinámica:

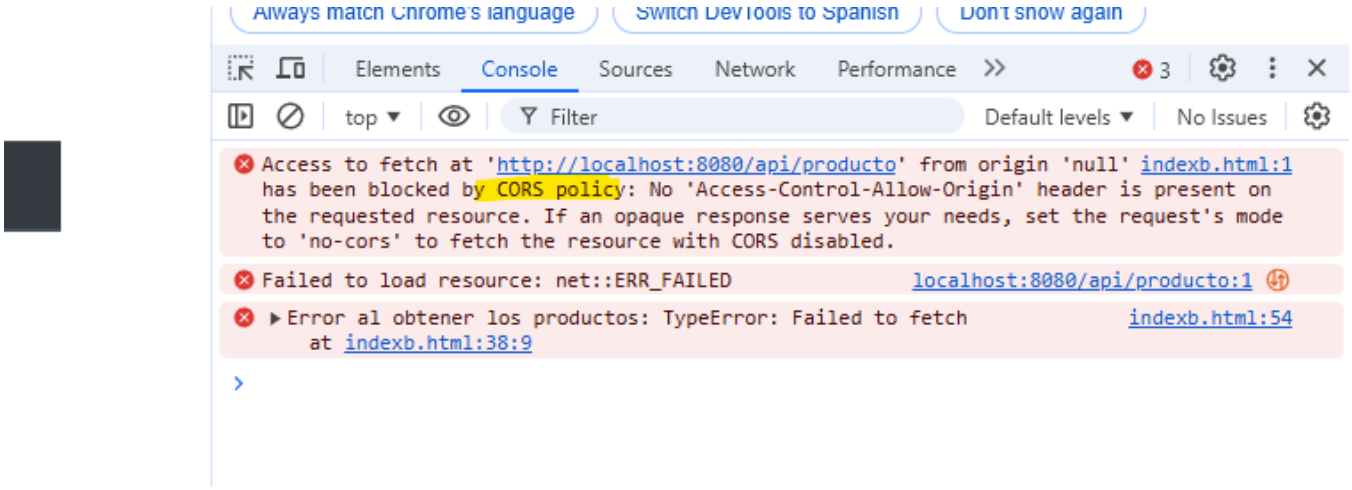
```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Lista de Productos</title>
    <!-- Enlace a Bootstrap CSS -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>
    <div class="container mt-5">
        <h1 class="mb-4">Productos</h1>
        <table class="table table-striped table-hover">
            <thead class="thead-dark">
                <tr>
                    <th>ID</th>
                    <th>Nombre</th>
                    <th>Descripción</th>
                    <th>Precio</th>
                    <th>Categoría</th>
                </tr>
            </thead>
```

```
            <tbody id="productos">

                <!-- Los productos se insertarán aquí -->

            </tbody>
        </table>
    </div>

    <!-- Enlace a jQuery y Bootstrap JS -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.4/dist/umd/popper.min.js"></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

    <script>
        fetch('http://localhost:8080/api/producto')
            .then(response => response.json())
            .then(data => {
                const productosTable = document.getElementById('productos');
                data.forEach(producto => {
                    const row = document.createElement('tr');
                    row.innerHTML = `
                        <td>${producto._id}</td>
                        <td>${producto.nombre}</td>
                        <td>${producto.descripcion}</td>
                        <td>${producto.precio}</td>
                        <td>${producto.categoria}</td>
                    `;
                    productosTable.appendChild(row);
                });
            })
            .catch(error => console.error('Error al obtener los productos:', error));
    </script>
</body>
</html>
```

El script, resaltado en rojo se hace lo siguiente:

- Se utiliza fetch para realizar una solicitud a la API en http://localhost:8080/api/producto.
- Se convierte la respuesta a formato JSON con response.json().
- Se recorre la lista de productos (data) y se crea una fila (tr) para cada producto.
- Cada fila se llena con las propiedades del producto (id, nombre, descripción, precio, categoría).
- Finalmente, cada fila se añade al cuerpo de la tabla (tbody) con appendChild.

Si recargamos el html vemos que no pasa nada y si abrimos la consola del navegador veremos que tenemos en error de CORS:



El error que estás viendo es un problema común relacionado con la política de CORS (Cross-Origin Resource Sharing). CORS es una política de seguridad que los navegadores implementan para restringir cómo los recursos en una página web pueden ser solicitados desde otro dominio.
En términos de CORS, un **origen** está compuesto por tres partes:

1. **El protocolo** (http, https)
2. **El dominio** (localhost, example.com, etc.)
3. **El puerto** (80, 8080, etc.)

Cuando el protocolo, el dominio o el puerto son diferentes entre la página web (el cliente) y la API (el servidor), se considera que están en orígenes diferentes, incluso si están en la misma máquina (localhost).

La mejor manera de manejar esto es configurar el servidor (en nuestro API) para permitir solicitudes desde cualquier origen. Usando un servidor Node.js con Express, podemos hacer esto de la siguiente manera:

1. Añadir en nuestro backend las librerías de cors:

   npm i -S cors

2. Comprobamos que se añadió en nuestro package.json

```
"dependencies": {
    "cors": "^2.8.5",
    "express": "^4.21.1",
    "mongodb": "^6.11.0",
    "mongoose": "^8.8.2"
  }
```

3. Añadir la librería en nuestro index.js

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

4. Añadir cors al app

```
app.use(cors());
```

Una vez añadido esto el error de cors desaparece.
En mi caso, tal como he implementado el controlador del endpoint:

```
        const productos = await productoService.obtenerProductos();
        res.status(200).send({ productos });
```

La lista de productos no viene directamente en el json sino que están bajo la palabra productos, como se puede ver aquí:

Es por esto que el script tal cual no me funciona porque tengo que hacer el recorrido foreach sobre la lista de productos, no sobre data. Analizad vuestro caso para ver la modificación que necesitáis, en mi caso ha sido la siguiente:

```
    <script>
        fetch('http://localhost:8080/api/producto')
            .then(response => response.json())
            .then(data => {
                console.log(data.productos);
                const productosTable = document.getElementById('productos');
                data.productos.forEach(producto => {
                    const row = document.createElement('tr');
                    row.innerHTML = `
                        <td>${producto._id}</td>
                        <td>${producto.nombre}</td>
                        <td>${producto.descripcion}</td>
                        <td>${producto.precio}</td>
                        <td>${producto.categoria}</td>
                    `;
                    productosTable.appendChild(row);
                });
            })
            .catch(error => console.error('Error al obtener los productos:', error));
    </script>
```

El resultado obtenido una vez recargado la página es el siguiente:

Productos

ID	Nombre	Descripción	Precio	Categoría
6744a2931e4011ce6e0718ec	Producto A	Este es el producto A	29.99	Categoría 1
6744e5e04660c3defc185872	Producto B	Este es el producto B	30.5	Categoría 1
6744e8f2e6bd52c1ad99ce4f	Producto C	Este es el producto C	18	Categoría 1

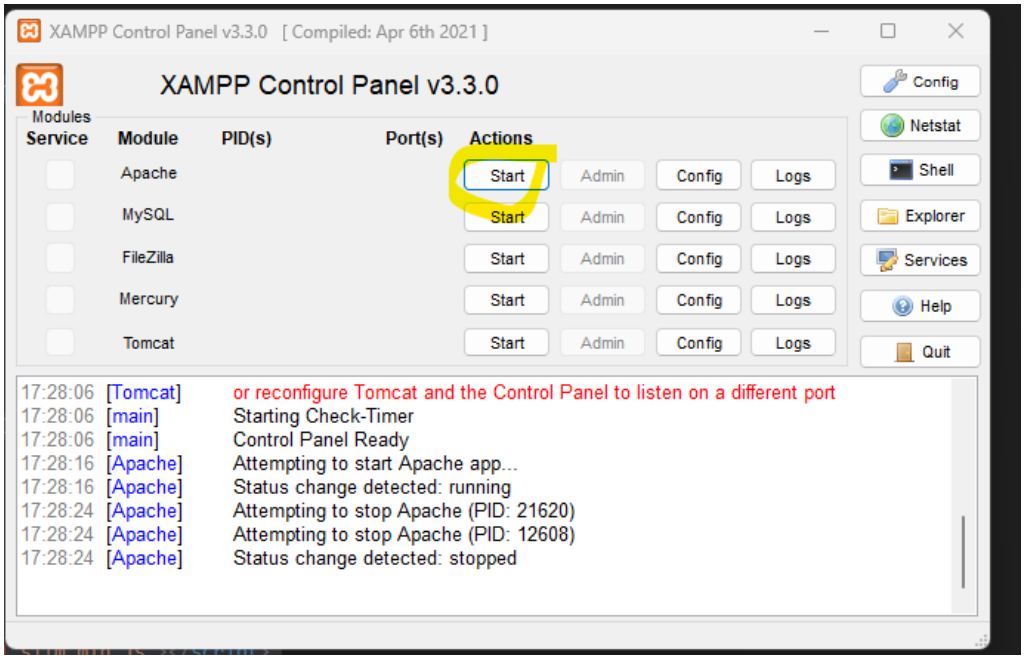
DESPLEGAR EL FRONTEND

Hasta ahora hemos ejecutado la aplicación frontend directamente desde nuestro ordenador, como un fichero html, ya que el navegador es capaz de interpretarlo. Si fuéramos a desplegar esta aplicación en producción, tenemos que alojarla en un servidor web, de modo que cuando en usuario solicite una URL desde el navegador, reciba nuestro html para que su navegador interprete y ejecute nuestra aplicación frontend.

Para ello, vamos a instalar Apache en nuestro ordenador, utilizando para el paquete xampp que incluye apache además de otras herramientas. XAMPP es una distribución de Apache fácil de instalar que contiene MariaDB, PHP y Perl. Es muy útil para desarrollar y probar aplicaciones web en un entorno local. A continuación, te proporciono instrucciones para instalar XAMPP en Windows:

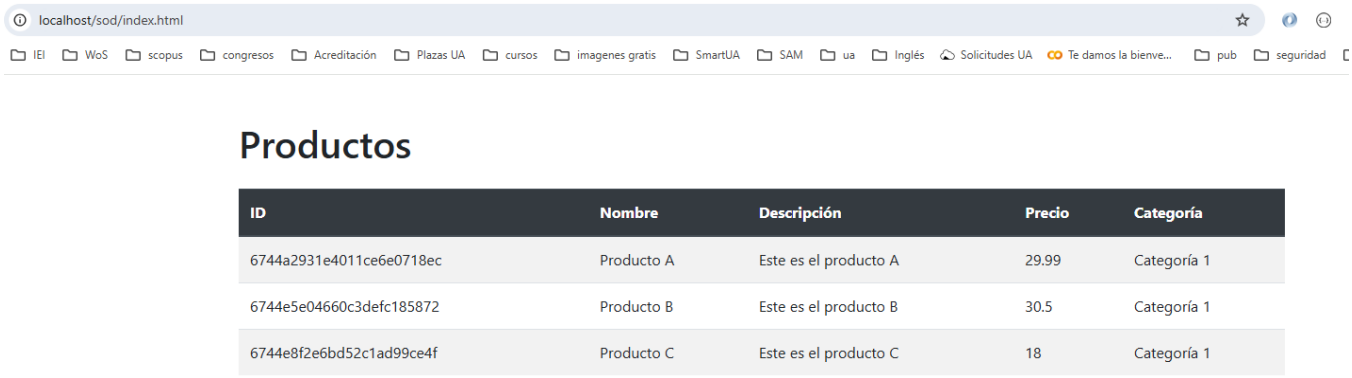
1. **Descargar XAMPP:**
   - Ve al sitio web oficial de XAMPP: https://www.apachefriends.org/es/index.html
   - Haz clic en el botón de descarga para Windows.
2. **Ejecutar el Instalador:**
   - Una vez que se haya descargado el archivo, haz doble clic en él para ejecutarlo.
   - Si aparece una advertencia de seguridad de Windows, haz clic en "Sí" para permitir la ejecución del instalador.
5. **Instalación:**
   - En la pantalla de bienvenida del instalador de XAMPP, haz clic en "Next".
   - Selecciona los componentes que deseas instalar. Por defecto, todos los componentes están seleccionados. Puedes desmarcar los que no necesites, pero es recomendable dejar la selección por defecto. Haz clic en "Next".
   - Elige la carpeta de destino donde deseas instalar XAMPP. Por defecto es C:\xampp. Haz clic en "Next".
   - Selecciona el idioma y haz clic en "Next".
   - Haz clic en "Next" nuevamente para iniciar la instalación.
6. **Finalización:**
   - Una vez completada la instalación, puedes optar por iniciar el Panel de Control de XAMPP marcando la casilla correspondiente y haciendo clic en "Finish".
   - El Panel de Control de XAMPP te permitirá iniciar y detener los servicios como Apache y MySQL.

Desde el panel de control iniciamos el servidor de apache:



Teniendo apache levantado podemos ver lo que devuelve http://localhost , al no poner puerto sería directamente al puerto 80 que es el predeterminado de http y es por el que escucha apache. En esta página vemos la página de inicio de XAMPP:

Recordemos que un servidor web está a la espera de solicitudes de los clientes de recursos concretos. Esos recursos que devuelve están en una carpeta concreta dentro de los servidores web, no es que busque cualquier directorio. La carpeta por defecto donde suelen estar los archivos que puede servir un servidor en htdocs.
Si entramos a la carpeta de instalación de XAMPP c://xampp veremos la carpeta htdocs que incluye por defecto varias carpetas incluida la dashboard con el html que nos muestra por defecto.
Dentro de esa carpeta vamos a crear una carpeta llamada sod y dentro copiaremos la aplicación frontend que hemos desarrollado.
Ahora si llamamos a la siguiente URL desde el navegador: http://localhost/sod/index.html el resultado debe ser nuestra aplicación:



Teniendo nuestro servidor en ejecución, un navegador desde otro ordenador podría acceder a nuestro servidor estando en nuestra red y utilizando la IP de nuestro ordenador.
Si quisiéramos poner nuestra aplicación en producción, debemos tener un ordenador conectado en una IP pública, accesible desde todas partes. De esta forma podrían acceder utilizando nuestra IP. Por otra parte, lo ideal sería comprar un dominio que se registre en los servidores DNS y de esta forma que accedan a nuestra aplicación utilizando una URL convencional.

Nota: Abrir un archivo HTML directamente desde tu ordenador y abrir el mismo archivo HTML a través de un servidor Apache en localhost tienen diferencias importantes en términos de cómo se procesan y se sirven los archivos.

Abrir un HTML directamente desde el ordenador

1. **Método de acceso**:
   - Simplemente haces doble clic en el archivo HTML o lo arrastras a un navegador.
   - La URL en el navegador se verá algo así como file:///C:/ruta/al/archivo/index.html.
2. **Contexto de ejecución**:
   - El archivo HTML se carga directamente desde el sistema de archivos del ordenador.
   - No hay servidor web involucrado en el proceso.
   - El navegador lee el archivo y renderiza el contenido.

Abrir un HTML a través de Apache en localhost

1. **Método de acceso**:
   - El archivo HTML es servido a través del servidor web Apache.
   - Accedes a la URL en el navegador como http://localhost/nombre-del-archivo.html o http://localhost/carpeta/nombre-del-archivo.html.
2. **Contexto de ejecución**:
   - El servidor Apache procesa la solicitud, localiza el archivo HTML en su estructura de directorios y lo envía al navegador.
   - Puedes tener configuraciones adicionales en Apache que afecten cómo se sirve el archivo (por ejemplo, configuraciones de seguridad, reescritura de URL, etc.).

EVALUACIÓN

- La entrega y presentación de esta la práctica 3 el 20 de diciembre.
- La entrega será solamente el envío del código de backend y frontend y el peso fundamental de la evaluación serán las preguntas formuladas en evaluación de la práctica.
- La evaluación se realizará el día 20 de diciembre en la clase de prácticas.