# DESENVOLVEMENTO DE INTERFACES WEB 22_23_Ord

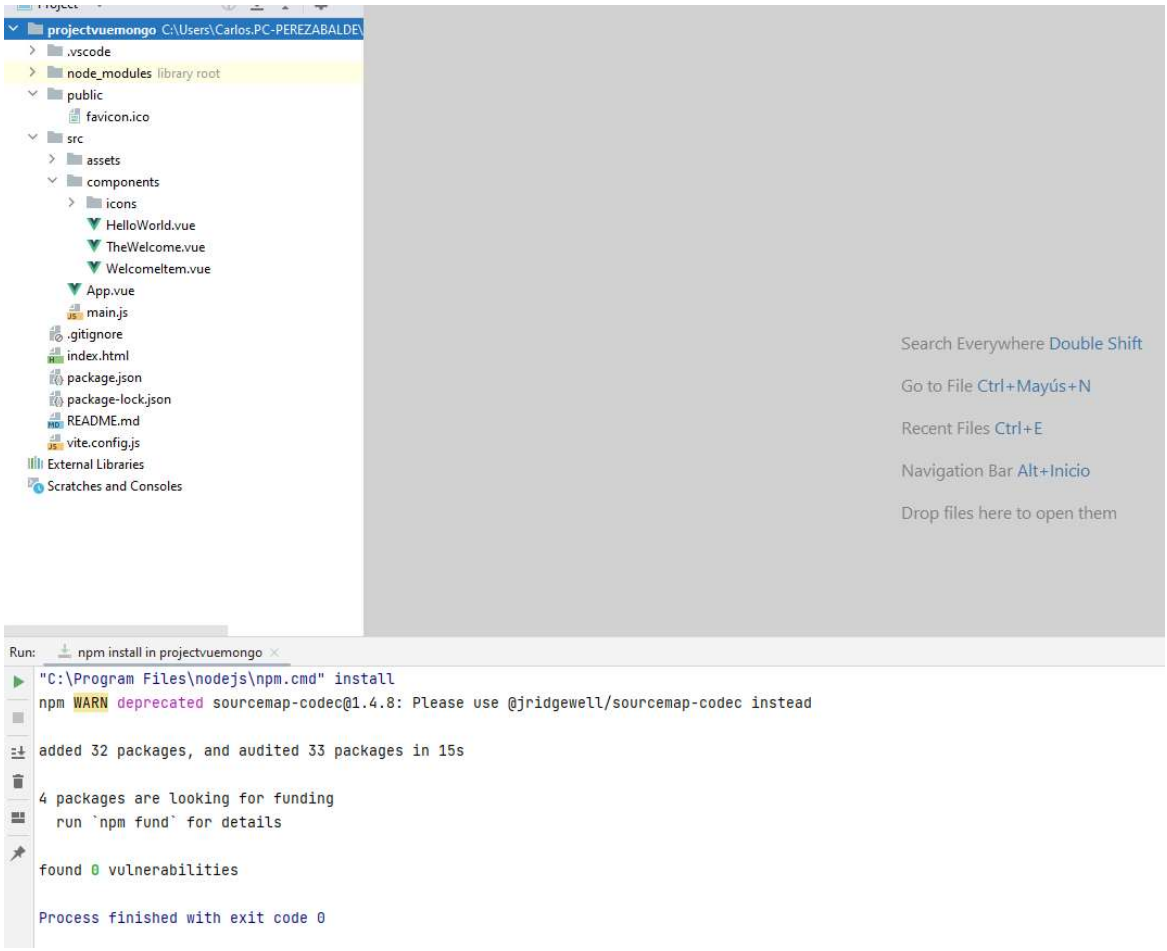


## 3. MEVN. MongoDB, Express, Vue y Node. Configuración del Servidor. ( I )

**El Stack MEVN,** tiene como objetivo crear aplicaciones web modernas con **Javascript** como principal lenguaje tanto el **Frontend** como en el **Backend** y la **Base de datos.**

**1. Preparación del entorno: configuración del Servidor**
Vamos a partir del hecho que tenemos instalado **Node.js.**.

Lanzamos un proyecto en uno de los IDE, instalando las dependencias **npm** que te solicita la propia herramienta.




**1.** Vamos a empezar con las configuraciones básicas con los **comandos siguientes para montar el servidor**:

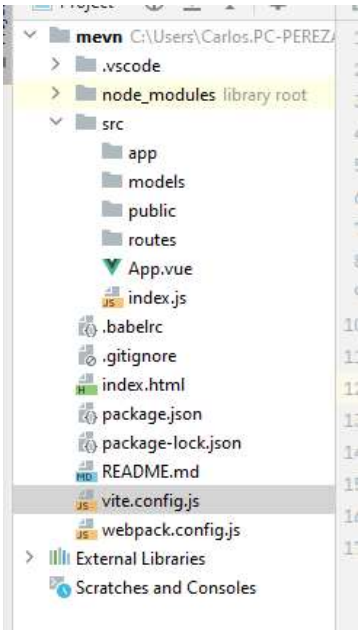
```
Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\Carlos.PC-PEREZABALDE\WebstormProjects\projectvuemongo> npm init --yes
Wrote to C:\Users\Carlos.PC-PEREZABALDE\WebstormProjects\projectvuemongo\package.json:

{
  "name": "projectvuemongo",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "dev": "vite",
```

Version Control | Run | TODO | Problems | Terminal | Services

**Iniciando un nuevo proyecto Node.js**

Creamos el fichero **.babelrc** en la **zona de los ficheros de configuración.**
Este fichero permite traducir el código **JS** de las últimas versiones a todo tipo de navegador y versión, en realidad la función lo hace el módulo **babel.**
También **webpack.config.js** para configurar diferentes opciones del proyecto **y facilita la generación del proyecto.**

A continuación renombramos dentro de **src,** el fichero **main.js** a **index.js.**
También dentro de src creamos dos carpetas más**:**
**-routes;** que nos indicará las diferentes rutas o urls del proyecto
**-models:** que albergará los diferentes **modelos de la base de datos** del proyecto dentro de mongodb
**-public:** siempre la vamos a enviar al navegador. **Contendrá todo el código vue.js transformado** para que lo entienda el navegador.
**-app:** donde estará todo el código del cliente y archivos front-end que luego se envía todo convertido a **public.**



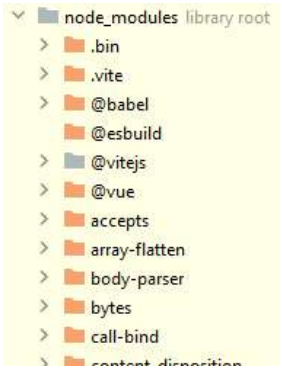

Vamos a empezar configurando el servidor. Para ello ejecutamos

```
LDE\WebstormProjects\mevn> npm install express
```

Este comando además de **instalar express lo agrega al package.json**

```
{
  "name": "mevn",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "express": "^4.18.2",
    "vue": "^3.2.45"
  },
```

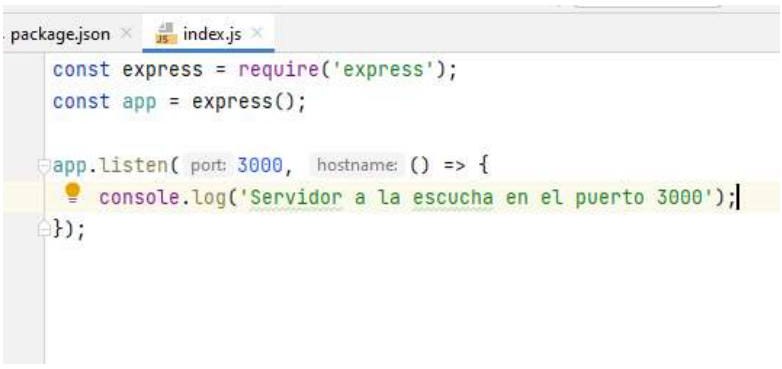
Además crea una carpeta llamada **node_modules** que contiene unas librerías de código que usa express y que **no hace falta tocar.**




Express es el **framework web** siendo una librería subyacente para un gran número de otros frameworks web de Node populares. Proporciona mecanismos para:

- Escritura de **manejadores de peticiones con diferentes HTTP en diferentes caminos URL (rutas).**
- Integración con **motores de renderización de "vistas"** para generar respuestas mediante la introducción de datos en plantillas.
- Establecer las rutas de las aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
- Añadir procesamiento de **peticiones "middleware"** adicional en cualquier punto dentro del manejo de la petición.

A continuación añadimos en **index.js:**




Básicamente lo que hemos hecho es preparar el servidor para que **escuche el puerto 3000,** luego cuando lo ponemos **en producción** podemos pasar al **puerto 80** habitual
Si todo ha ido bien cuando lo lanzamos mandará el mensaje que se observa en la imagen a continuación.


En el navegador:


```
Cannot GET /
```

Aunque se muestre un mensaje de error no es que esté mal codificado, simplemente nos dice que no tiene ninguna vista que abrir ni ruta establecida.

El siguiente código nos permite utilizar el código que nos interese no solo el 3000 por defecto.

Declarar **el puerto como una variable** es una buena estrategia para que cuando pasemos a producción simplemente tengamos que cambiar aquí el puerto 80.


Si cambiamos el puerto por 5000. El mensaje es el mismo y el servidor funciona pero en un puerto distinto.


```
Cannot GET /
```

Sin embargo, en un servidor en internet el número se cambia por el atributo **process.env.PORT**, es decir el puerto habitual que nos da el sistema.

En caso contrario, usaría el 3000 si ponemos el código de la siguiente forma:

```
const express = require('express');
const app = express();
// Configuraciones de express
app.set('port', process.env.PORT || 3000);
```

De una manera u otra ya está preparado para poner en producción o en desarrollo.

Avanzando con la configuración del servidor.
Vamos a establecer la ruta hacia el **directorio el cual contendrá todos los ficheros estáticos o public** que el servidor entrega el **cliente para que este (el navegador) los utilice**, es decir, *ficheros html, ficheros css, imágenes....*
Lo haremos de la forma siguiente:

```
ckage.json  index.js
const express = require('express');
const app = express();
// Configuraciones de express
app.set('port', process.env.PORT || 3000);

//middlewares

//Rutas


//Ficheros estáticos
app.use(express.static( root: __dirname + '/public'))

// servidor está escuchando
app.listen(app.get('port'),  hostname: () => {
    console.log('Servidor a la escucha en el puerto', app.get('port'));
});
```

Dentro de public creamos un fichero HTML sencillo, **index.html.** Podemos aprovechar el index.html que creo Webstorm o codificamos uno, pero dentro de public.


El resultado sería:


En **public podemos incluir rutas a ficheros javascript o css que enlacemos desde el fichero inde**x u otro que definiremos en las rutas (routes)

A continuación vamos a instalar **nodemon,** este módulo es un *demonio o daemon* que nos evita tener que estar relanzando el servidor cada vez que hagamos cambios en el proyecto, mostrándonos los resultados de forma rápida, ya que él se encarga de reiniciar el servidor.

```
E\WebstormProjects\mevn> npm install nodemon
```

Hacemos una pequeña modificación en **package.json,** para ello necesitamos arrancar el servidor con el siguiente comando

```
WebstormProjects\mevn> npm run dev
```

Cada vez que hagamos un cambio ya solo necesitamos refrescar la página no hace falta ya que daemon se encarga de reiniciar el servidor si detecta algún cambio, en el index, en el puerto a usar, etc...

**Nota.-** puede que aparezca un mensaje ***missing script dev***, en dicho caso vamos al fichero package.json y añadimos la siguiente línea en el apartado de ***scripts:***

```
"scripts": {
  "dev": "nodemon src/index.js",
```

Vamos a configuar la parte de **middlewares.** Un ***middleware*** es un software que **permite que las aplicaciones de un sistema se comuniquen entre sí** y, también, con otros paquetes de software, con el sistema operativo y con elementos hardware.
Uno de ellos es el módulo **morgan. Morgan** es un *middleware* **para la captura de solicitudes HTTP para Node.js para su posterior registro y seguimiento entre el servidor y el navegador.**

```
:\WebstormProjects\mevn> npm install morgan
```

Y configuramos **index.js**

```
const express = require('express');
const app = express();
const morgan = require('morgan');
// Configuraciones de express
app.set('port', process.env.PORT || 3000);

//middlewares
app.use(morgan( format: 'dev'));
app.use(express.json());
//Rutas
```

En este caso el uso de **morgan** nos mostrará más información por terminal, gracias al parámetro **dev.** Si refrescamos la página vemos que:

```
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Servidor a la escucha en el puerto 3000
GET / 304 8.170 ms - -
```

Ha habido una petición GET que tardo en este caso 8.17 ms.
También hacemos que express cargue el módulo **json** para que cada vez que el servidor reciba un fichero **json** desde el navegador pueda entenderlo y procesarlo. Ahora a viene instalada con **express**.
Antes venía separado de **express,** en un módulo *body-parser* para ahora ya está integrado.

Dentro de **routes** vamos a crear el fichero **tareas.js** en el cual añadimos el código siguiente:

Este fichero configurará y definir las diferentes rutas. Cuando el servidor reciba un petición de una determinada **req**, enviará la una respuesta **res.**
Luego los exportamos para usarlo en otros lugares de la aplicación.
Evidentemente el servidor deberá incorporar o conocer el fichero, con lo que hacemos lo siguiente en **index.js.**

De esta forma cuando el servidor necesite conocer una ruta, en nuestro siempre será tareas, sabrá donde tiene que ir a buscarla**.**
**Se pueden poner más pero la aplicación ser de una sola página.**

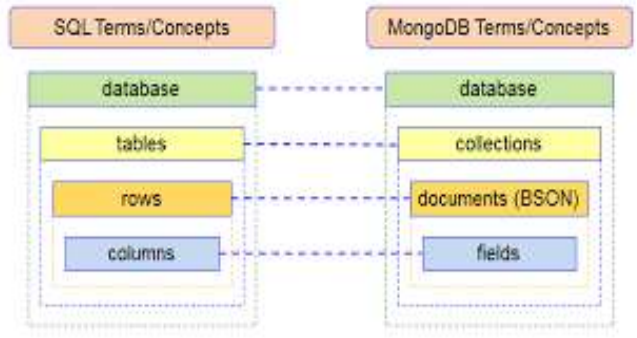
En el navegador si ponemos la ruta:

También hay que observar que tenemos un *get,* pero también habrá un *put, post o un delete,* y todo eso nos permite ir a por la siguiente y última fase de la configuración del servidor: conectarse a una Base de Datos, en nuestro caso **mongodb**

**MongoDB** es una base de datos NoSQL, es decir, no se basa en el esquema E/R tradicional. Con el aumento en el almacenamiento de datos los DBA se han encontrado que los esquemas basados en entidades relacionales se están convirtiendo en cuellos de botella en el rendimiento

de las bases de datos.
Ello ha provocado que se vuelva (ya existía en el pasado) al almacenamiento en ficheros como hacen los gestores de bases de datos NoSQL.
Las palabras clave en mongdb son dos:
**- colecciones:** son un conjunto de documentos, vienen a ser como las tablas en MySQL o similares
**- documentos:** son ficheros, muy parecido a ficheros json, que contienen un determinado registro, es decir, sería como una fila de una tabla.

Más sobre mongodb **aquí.**




| Operación | SQL | MongoDB |
|---|---|---|
| Creación base de datos | **CREATE DATABASE** nombreBD; | **USE** nombreBD |
| Creación tabla (en SQL) / Colección (en MongoDB) | **CREATE TABLE** nombreTabla ( atributo1 tipo_de_dato restricciones <, atributo2...>); | **db.createCollection**("nombreColección") |
| Creación índice | **CREATE INDEX** nombreIndice **ON** nombreTabla (atributo1 <, atributo2... >); | **db.nombreColeccion.ensureIndex**( {"atributo": 1}) |
| Destrucción tabla (en SQL) / Colección (en MongoDB) | **DROP TABLE** nombreTabla; | **db.nombreColeccion.drop()** |

Descargamos Mongodb en Windows (Version Community). Puede que tengamos que cargar la ruta del comando **mongod** en el PATH del sistema como variable de entorno. (c:\archivos de programa\mongodb\server\6.0\bin). También necesita que dentro de la carpeta **data existe otra db**

**Creación de un documento:**

Continuemos con la configuración del servidor, concretamente con la base de datos. Para eso vamos a utilizar un módulo llamado **mongoose es una librería para Node.js que nos permite escribir consultas para una base de datos de MongooDB,** con características como validaciones, construcción de queries, middlewares, conversión de tipos y algunas otras, que enriquecen la funcionalidad de la base de datos.

Probamos ejecutando **mongod** en un terminal:

y más abajo aparece que ya está a la escucha.

Ahora desde **index.js llamamos a la librería para su uso.**

Lanzamos el servidor:

Vemos que la base de datos está conectada.

El paso siguiente es crear los modelos correspondientes de los datos que queremos manejar en la base de datos. Antes creamos la carpeta **modelos.**
Este directorio es donde se definirán los modelos que creamos. Creamos un fichero llamado **Tarea.js** (mayúsculas para no confundirnos) y codificamos lo siguiente.

Básicamente lo que hacemos es crear un **Schema o esquema de los datos** que queremos manejar en MongoDB, en este caso, ***Tareas*** y como siempre con cualquier objeto, mejor dicho, modelo en este caso hay **exportarlo para utilizarlo**.
Para no complicarse al le llamamos de la misma forma que el modelo.

Ahora tenemos que ir al directorio *routes* y establecer la configuración necesaria para acceder a *Tarea*s, es decir, que el servidor sepa donde tiene que buscar el módulo. Se hace de esta forma:

Veamos. Como ya comentamos en otros proyectos las peticiones en JS son asíncronas. Para evitar utilizar los **promises** de JS.
Los métodos **async/await** funcionan igual pero evitan escribir bastante código.
Una función **async** puede contener una expresión **await,** la cual pausa la ejecución de la función asíncrona y espera la resolución de la petición pasada. Cuando dicha resolución llega, se reanuda la ejecución de la función **async** devolviendo el valor o solicitud en cuestión.

Por otro lado está la función **find()** que es propia de MongoDB y que funciona como el **select** de SQL**,** es decir, devuelve una lista de valores.
Pues bien, cuando se ejecuta el **get,** la función **await** para la ejecución del método hasta que la BD le devuelve los datos solicitados, en este caso un array vacío, que almacena en la variable tareas.
Veamos el resultado por pantalla. Si no se ve, a veces, conviene reiniciar el servidor.

Observemos en terminal la petición:

Ahora iremos a por la ruta POST, que funciona como GET, pero en este caso almacenamos datos en la base de datos, a través de **request** es la **información que envía el navegador al servidor.**
Crearíamos una tarea, a través de una instancia de la clase Tareas, tal como se hace JS o en JAVA. El navegador **no puede enviar métodos POST** directamente sino a **través de formularios.** Como no tenemos aún el frontend utilizaremos un plugin de **Postman.**

**Postman** es una plataforma que posibilita y facilita la creación y el uso de APIs. Se puede usar, por ejemplo, para obtener información sobre las respuestas HTTP, en diferentes métodos, que realicemos a APIs de diferentes temáticas. En nuestro caso es para comprobar métodos de POST, GET, DELETE y PUT.

Vamos a **https://www.postman.com/downloads/** y descargamos **el Postman Desktop Agent.**

Al lanzarlo nos aparece la ventana siguiente desde la cual podemos hacer peticiones GET.

¿Como se usa? Pues simplemente indicando al lado de GET la dirección URL, por ejemplo,

Nos aparece vacío.

Vamos ahora con una petición POST. Primerio añadimos el siguiente código en el fichero **tareas.js** de **routes**

Y ahora desde postman lanzamos **una petición POST** en este caso

Parece que se queda pensando pero si vamos al terminal, vemos que ha creado un nuevo Objeto. Este objeto es **una nueva tarea** pero como no tenemos formulario **porque aún no codificamos** el frontend pues no muestra nada, solo el id del objeto.

Evidentemente todo esta vacío.

Ahora en Postman vamos a crear una tarea de la forma siguiente **clickeando en body:**

Si falla con ***x-www-form-urlencoded,*** entonces usaremos ***raw*** **con datos de tipo JSON.**

Modificamos el fichero **tareas.js de routes para testear que fue recibido.**

**Reinicimos el servidor,** porque a veces queda en memora la ejecución anterior, y le damos **send, pero antes cambiamos en Header a lo siguiente:**

Además en consola vemos el objeto creado con los datos.

Vemos el **objeto creado** y los datos en formato **JSON.** Por cierto, el **id es creado por MongoDB** con el formato utilizado por este gestor de BD.

Evidentemente, la finalidad del método POST no es mostrar en terminal sino guardar en una base de datos. Para ello el **código de tareas.js** lo modificamos de la siguiente manera.

Observamos el método **await** para que **el cliente espere la respuesta del servidor y el método status para que nos indique el estado de la ejecución.**
Vamos ahora al Postman y ejecutamos el POST y luego el GET para ver el resultado del POST.

Y el método **GET** vemos el objeto creado.

Y en el **navegador:**

Vamos a crear **otra tarea:**

El método **GET con los dos objetos:**

Y en el **navegador:**

Vamos a actualizar **PUT,** en este caso **necesitamos el id** de la tarea que creó MondoDB**, y que es el parámetros o *params,*** y también el ***req.body*** que son los datos a actualizar. El código sería así:

Vamos a modificar la descripción de la tarea de trámites de Hacienda diciendo el número de declaraciones trimestales a presentar**.** Para ello copiamos el id de la tarea: *63d53ea7f2d09ec16e108065*

Indicamos el método **PUT:**

Y en **GET:**

Y en el **navegador:**

Nos queda **eliminar o DEL, que también utilizaremos el id** y, en este caso, eliminaremos los trámites ante Hacienda

***Nota.***- *usad IdandRemove y NO IdandDelete*

Vamos a Postman**, sin olvidarse de poner el id.**

El método **GET** con solo una tarea.

Y el **navegador.**

Ya tenemos el servidor funcionando, ahora nos vamos a la 2º parte que es la codificación de la app con Vue.js, es decir, el frontend de la aplicación.

Última modificación: Martes, 7 de Febreiro de 2023, 11:35

◄ 2. API REST con Vue.js

Ir a...

4. MEVN. MongoDB, Express, Vue y Node. Configuración del Frontend. ( II ) ►