

DWEC – Javascript Web Cliente.

Pasos crear una API con node.js y express	1
Añadir rutas:	4
Guardar datos en BD de MySQL.....	5
Detener una aplicación Express en Node.js	7
Para no tener que reiniciar el servidor con cada cambio:.....	9
Ejemplo de las tablas:	10
El fichero app.js quedaría:	11
Vamos a poner las rutas en distintos ficheros de rutas:	13
App.js	13
Rutas-ciudades.js	14
Sacamos la lógica a ficheros controladores:	16
Sacar la conexión a mysql en archivo aparte	18
Servir archivos estáticos	21
CORS y Fetch desde otro equipo	23
No es necesario abrir el Puerto de express en firewall.....	26
Actuaciones con el puerto de MySql	28

Pasos crear una API con node.js y express

Ejemplo de cómo estructurar una API utilizando Node.js y Express:

```
api/
├── controllers/
│   ├── userController.js
│   └── ...
├── models/
│   ├── user.js
│   └── ...
├── routes/
│   ├── userRoutes.js
│   └── ...
└── app.js
```

En esta estructura, tenemos las siguientes carpetas:

controllers: Esta carpeta contendrá los controladores de la API. Cada controlador se encargará de manejar la lógica de negocio relacionada con un modelo específico.

models: Esta carpeta contendrá los modelos de la API. Cada modelo representa una entidad en la base de datos y define su estructura y comportamiento.

routes: Esta carpeta contendrá las definiciones de las rutas de la API. Cada archivo de ruta se encargará de manejar las solicitudes que lleguen a una ruta específica.

app.js: Este archivo será el punto de entrada de la aplicación. Aquí configurarás y montarás todas las partes de la API, como los middleware y las rutas.

Veamos los pasos desde VSC:

1. Inicializa tu proyecto:

Abre tu terminal y ejecuta los siguientes comandos para crear un nuevo proyecto y entrar en el directorio:

```
bash
mkdir mi-api
cd mi-api
npm init -y
```

2. Instala Express:

Ejecuta el siguiente comando para instalar Express:

```
bash
npm install express
```

3 Crea el archivo principal e la aplicación (app.js)

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

// Crear una instancia de Express
const app = express();

// Habilitar CORS para todas las rutas
app.use(cors());


// Resto de tu configuración de Express y rutas aquí...

// Puerto en el que escuchará el servidor
const puerto = 3000;

// Iniciar el servidor y escuchar en el puerto especificado
app.listen(puerto, () => {
  console.log(`Servidor escuchando en el puerto ${puerto}`);
});
```

Importar módulos: Importamos los módulos necesarios `express` y `cors` utilizando `require`.


javascript

 Copy code

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

Crear instancia de Express: Creamos una instancia de Express llamada `app`. Express es un framework web para Node.js que facilita la creación de aplicaciones web y APIs.


javascript

 Copy code

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

Habilitar CORS: Utilizamos el middleware `cors` para habilitar CORS en todas las rutas de nuestra aplicación. Esto permite solicitudes desde cualquier origen.


javascript

 Copy code

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

Configuración de rutas: Aquí es donde agregarías las rutas específicas de tu aplicación. Esto puede incluir rutas para manejar solicitudes GET, POST, PUT, DELETE, etc.

javascript

 Copy code

```
// Resto de tu configuración de Express y rutas aquí...
```

Deberías agregar tus rutas y lógica de manejo de solicitudes aquí, dependiendo de los endpoints y la funcionalidad que necesites en tu aplicación.

Iniciar el servidor: Finalmente, iniciamos el servidor y lo hacemos escuchar en un puerto específico. En este caso, el servidor escuchará en el puerto 3000.

```
javascript Copy code  
  
const puerto = 3000;  
app.listen(puerto, () => {  
  console.log(`Servidor escuchando en el puerto ${puerto}`)  
});
```

Cuando ejecutas tu aplicación, el servidor comenzará a escuchar en el puerto especificado, y verás el mensaje en la consola indicando que el servidor está en funcionamiento.

Añadir rutas:

```
// Resto de tu configuración de Express y rutas aquí...  
// Middleware para parsear el cuerpo de las peticiones en formato JSON  
app.use(express.json());  
  
// Ruta de ejemplo  
app.get('/', (req, res) => {  
  res.send('¡Bienvenido a mi API!');  
});  
  
// Ruta para obtener datos  
app.get('/api/datos', (req, res) => {  
  // Simulación de datos  
  const datos = {  
    mensaje: 'Estos son tus datos',  
    usuario: 'ejemplo',  
    edad: 25  
  };  
  
  res.json(datos);  
});  
  
// Ruta para enviar datos  
app.post('/api/enviar', (req, res) => {  
  const { mensaje } = req.body;  
  
  // Puedes realizar operaciones con los datos recibidos  
  // En este caso, simplemente los devolvemos
```

```
res.json({ recibido: true, mensaje });  
});
```

Guardar datos en BD de MySQL

Para guardar datos en una base de datos MySQL desde tu aplicación Node.js con Express, primero necesitarás instalar el paquete `mysql` para Node.js. Puedes hacerlo ejecutando el siguiente comando:

```
bash
```

[Copy code](#)

```
npm install mysql
```

Luego, puedes modificar tu código para conectarte a la base de datos y guardar/recuperar datos. A continuación, te presento un ejemplo básico utilizando MySQL:

1. Instala el paquete MySQL:

```
bash
```

[Copy code](#)

```
npm install mysql
```

2. Modifica tu archivo `index.js`:

```
javascript
```

[Copy code](#)

```
const express = require('express');  
const mysql = require('mysql');  
const app = express();  
const port = 3000;  
  
// Middleware para parsear el cuerpo de las peticiones en formato JSON  
app.use(express.json());
```

El `index.js` completo queda así:

```
const express = require('express');  
const mysql = require('mysql');  
const app = express();  
const port = 3000;  
  
// Middleware para parsear el cuerpo de las peticiones en formato JSON  
app.use(express.json());  
  
// Configuración de la conexión a la base de datos
```

```
const db = mysql.createConnection({
  host: 'localhost',
  user: 'tu_usuario_mysql',
  password: 'tu_contraseña_mysql',
  database: 'tu_base_de_datos'
});

// Conexión a la base de datos
db.connect((err) => {
  if (err) {
    console.error('Error de conexión a la base de datos:', err);
  } else {
    console.log('Conexión exitosa a la base de datos');
  }
});

// Ruta para obtener datos desde la base de datos
app.get('/api/datos', (req, res) => {
  // Consulta a la base de datos
  db.query('SELECT * FROM datos', (err, resultados) => {
    if (err) {
      console.error('Error al obtener datos desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json(resultados);
    }
  });
});

// Ruta para enviar datos y guardar en la base de datos
app.post('/api/enviar', (req, res) => {
  const { mensaje } = req.body;

  // Insertar datos en la base de datos
  db.query('INSERT INTO datos (mensaje) VALUES (?)', [mensaje], (err, resultado) => {
    if (err) {
      console.error('Error al guardar datos en la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json({ recibido: true, mensaje, id: resultado.insertId });
    }
  });
});

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

3. Crea una tabla en tu base de datos:

Antes de ejecutar tu aplicación, asegúrate de tener una base de datos creada y una tabla para almacenar los datos. Puedes ejecutar la siguiente consulta SQL en tu cliente MySQL o herramienta de administración para crear una tabla de ejemplo:

```
sql Copy code  
  
CREATE TABLE datos (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  mensaje VARCHAR(255) NOT NULL  
);
```

4. Ejecuta tu aplicación:


Inicia tu servidor con `node index.js`, y ahora debería guardar y recuperar datos desde la base de datos MySQL.

Este es un ejemplo básico, y en un entorno de producción, se recomienda manejar las conexiones y errores de manera más robusta, y considerar medidas de seguridad como el uso de consultas preparadas para prevenir inyecciones SQL. Además, asegúrate de proteger la información de conexión a la base de datos.

Detener una aplicación Express en Node.js

Para detener una aplicación Express en Node.js, puedes utilizar el método `close()` del servidor que se crea al llamar a `listen()` en tu aplicación Express. Aquí hay un ejemplo de cómo hacerlo:

javascript

 Copy code

```
const express = require('express');
const app = express();
const port = 3000;

// Configura tus rutas y middleware aquí

const server = app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});

// Para detener la aplicación
// Puedes llamar a server.close() cuando sea necesario, por ejemplo, en un script

// Ejemplo de detener la aplicación después de 5 segundos
setTimeout(() => {
  console.log('Deteniendo la aplicación...');
  server.close();
}, 5000);
```

En este ejemplo, la aplicación Express se inicia llamando a `app.listen()`, y el servidor devuelto se almacena en la variable `server`. Luego, puedes llamar a `server.close()` para detener la aplicación.

Ten en cuenta que si estás ejecutando la aplicación dentro de un script o aplicación más grande, es posible que necesites gestionar la lógica de cierre de manera más robusta. Por ejemplo, podrías usar señales de sistema operativo para detectar solicitudes de cierre y cerrar la aplicación de manera controlada. Aquí hay un ejemplo utilizando `process.on` para manejar la señal `SIGINT` (Ctrl+C en la consola):

javascript

Copy code

```
const express = require('express');
const app = express();
const port = 3000;

// Configura tus rutas y middleware aquí

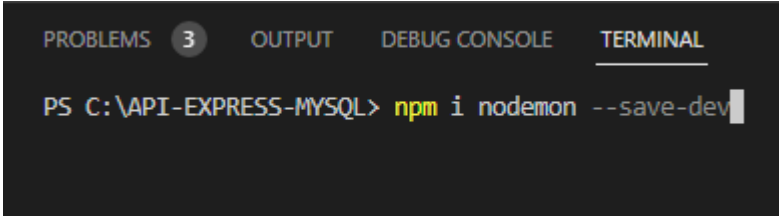
const server = app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});

// Manejar la señal SIGINT (Ctrl+C)
process.on('SIGINT', () => {
  console.log('Deteniendo la aplicación...');
  server.close(() => {
    console.log('La aplicación se ha detenido correctamente.');
```

Este enfoque te permite manejar de manera adecuada el cierre de la aplicación cuando se recibe la señal SIGINT.

Para no tener que reiniciar el servidor con cada cambio:

Instalamos (abreviamos con npm i) el módulo **nodemon** con la opción `--save-dev` para que lo instale como “una dependencia de desarrollo” y así solo se tenga en cuenta durante el desarrollo de la app, y no cuando esté en producción.



```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\API-EXPRESS-MYSQL> npm i nodemon --save-dev
```

Estos cambios los veremos en el archivo package.json

En `package.json` realizamos otros cambios:

```
{
  "name": "api-express-mysql",
  "version": "1.0.0",
  "description": "",
  // cambiamos main por el nombre del archivo de entrada (server.js)
  "main": "server.js",
  "scripts": {
    // "test": "echo \"Error: no test specified\" && exit 1"

    //Dentro de scripts ponemos el nombre start(podría ser otro) y el
    // nombre del archivo de entrada (server.js)
    "start": "nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1"
  }
}
```

Añadimos en “scripts” la línea:

“start”: “nodemon server.js”

Podemos poner otro nombre distinto a start, pero es el que nos va a servir para ejecutar el servidor con el comando: `npm run start`

```
PS C:\API-EXPRESS-MYSQL>
PS C:\API-EXPRESS-MYSQL> npm run start

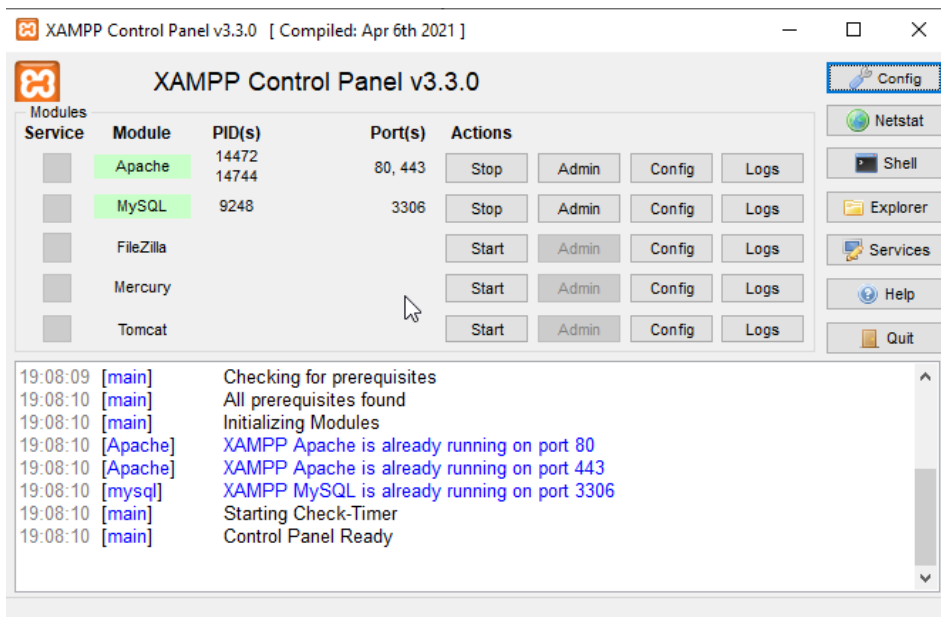
> api-express-mysql@1.0.0 start
> nodemon server.js

[nodemon] 2.0.19
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Servidor corriendo en puerto 9000
```

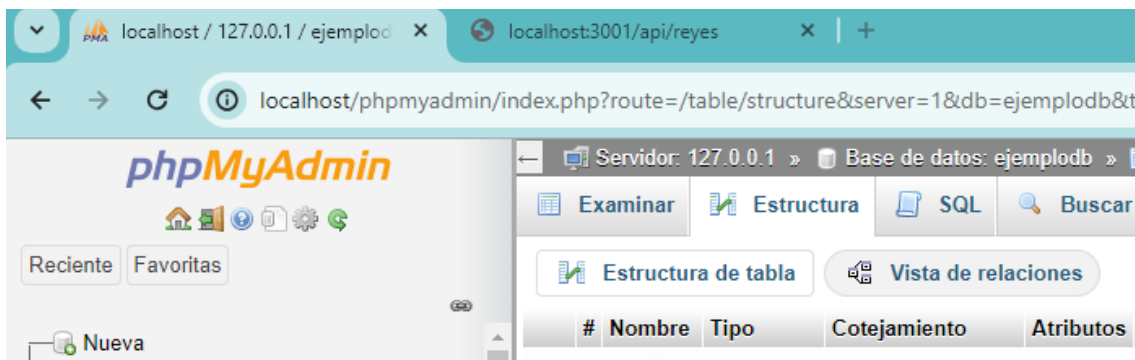
A partir de ahora cada vez que guardamos cambios (Ctrl+S) en el fichero .js, el servidor se reinicia solo.

Ejemplo de las tablas:

Tenemos XAMPP e con Apache y MySQL arrancados



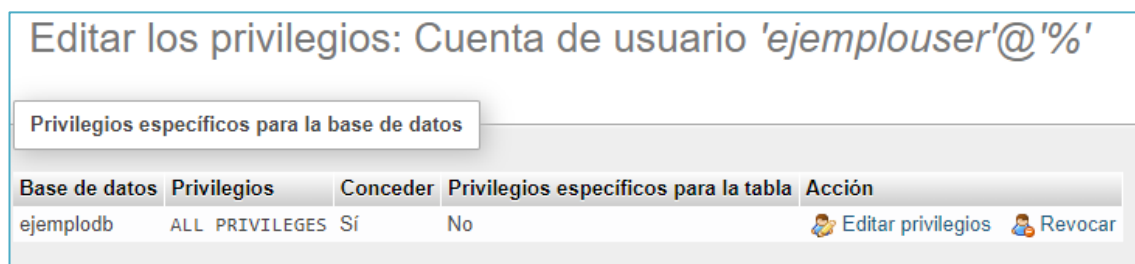
Podemos utilizar phpMyAdmin para administrar las tablas SQL, y los usuarios que tendrán acceso.



Es importante señalar que deberemos tener un usuario (distinto de root o del administrador MySQL) para acceder a la API.

Lo creamos en phpMyAdmin y asignamos los permisos a la base de datos de nuestra API.

Para nuestro ejemplo usaremos las tablas de la BD “ejemplodb” (ciudades, cuencas, embalses, paises, reyes)



En este caso tenemos el usuario “ejemplouser” con todos los privilegios para la base de datos “ejemplodb”. Además es accesible desde cualquier IP -> ‘ejemplouser@%’

El fichero app.js quedaría:

```

const express = require('express'); // se instala con: npm i express
const mysql = require('mysql'); // se instala con: npm i mysql
const app = express(); // esta es la aplicación que iniciamos
const port = 3000; // puerto para express
  
```

```
//EL PUERTO DEL SERVIDOR
// vamos a "setear" a través del método set de nuestra aplicación: ponemos la propiedad
//del puerto (port), y el valor: 3000
//app.set('port', 3000);
//mejoramos la línea anterior para que pueda usar el puerto que asigne el HOSTING cuando
//esté en producción, o el 3000.
app.set('port', process.env.PORT || port); // 'port' es una propiedad de app.set

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(express.json()); //middleware para JSON

// Configuración de la conexión a la base de datos
const db = mysql.createConnection({
  //propiedad host donde se ubica la base de datos Mysql
  host: '127.0.0.1',
  // port: 3306,      // se puede cambiar el puerto de mysql

  //usuario base de datos (por defecto sería root, pero hay que evitarlo)
  user: 'ejemplouser',
  //contraseña del usuario en mysql
  password: '123456',
  //nombre del esquema al que nos vamos a conectar
  database: 'ejemplodb'
});

// Conexión a la base de datos
db.connect((err) => {
  if (err) {
    console.error('Error de conexión a la base de datos:', err);
  } else {
    console.log('Conexión exitosa a la base de datos');
  }
});

// Ruta para obtener datos desde la base de datos
app.get('/api/datos', (req, res) => {
  // Consulta a la base de datos
  db.query('SELECT * FROM paises', (err, resultados) => {
    if (err) {
      console.error('Error al obtener datos desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json(resultados);
    }
  });
});

// Ruta para enviar datos y guardar en la base de datos
app.post('/api/enviar', (req, res) => {
  const { mensaje } = req.body;

  // Insertar datos en la base de datos
```

```
db.query('INSERT INTO datos (mensaje) VALUES (?)', [mensaje], (err, resultado) => {
  if (err) {
    console.error('Error al guardar datos en la base de datos:', err);
    res.status(500).json({ error: 'Error interno del servidor' });
  } else {
    res.json({ recibido: true, mensaje, id: resultado.insertId });
  }
});
});

// Inicia el servidor
// Hacemos que el servidor empiece a escuchar en el puerto 9000 (por defecto en express)
// se hace con el método listen pasándole el número de puerto y una función
// que nos diga si está funcionando
/*
// versión antigua
app.listen(app.get('port'), ()=>{
  console.log("Servidor corriendo en puerto", app.get('port'));
})
*/
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

Vamos a poner las rutas en distintos ficheros de rutas:

Quedando de momento 2 archivos:

App.js

```
//requiero rutas (pongo un nombre cualquiera) que he exportado en routes.js
const rutasCiudades= require('./routes/rutas_ciudades');//no necesita extensión .js
//Ahora el servidor tiene un link a las rutas de rutas-ciudades.js
//necesita el require de rutas exportado

const express = require('express'); // se instala con: npm i express

const app = express(); // esta es la aplicación que iniciamos
const port = 3000; // puerto para express

//EL PUERTO DEL SERVIDOR
// vamo a "setear" a través del método set de nuestra aplicación: ponemos la propiedad
//del puerto (port), y el valor: 3000
//app.set('port', 3000);
//mejoramos la línea anterior para que pueda usar el puerto que asigne el HOSTING cuando
//esté en producción, o el 3000.
app.set('port', process.env.PORT || port); // 'port' es una propiedad de app.set

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(express.json()); //middleware para JSON
```

```
// Middleware para parsear el cuerpo de las peticiones en formato JSON
//app.use(bodyParser.json());

app.use('/ciudades', rutasCiudades); //así /ciudades es la raíz en rutas_ciudades.js

// Inicia el servidor
// Hacemos que el servidor empiece a escuchar en el puerto 9000 (por defecto en
express)
// se hace con el método listen pasándole el número de puerto y una función
// que nos diga si está funcionando
/*
// versión antigua
app.listen(app.get('port'), ()=>{
  console.log("Servidor corriendo en puerto", app.get('port'));
})
*/
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

Rutas-ciudades.js

```
// fichero de rutas: rutas_ciudades.js
//creo la constante para utilizar el módulo expresss que es requerido
const express= require('express');

// constante para utilizar el método Router() que se necesita
// y que debemos exportar
const rutasCiudades = express.Router();

// Configuración de la conexión a la base de datos
const mysql = require('mysql'); // se instala con: npm i mysql
const db = mysql.createConnection({
  //propiedad host donde se ubica la base de datos Mysql
  host: '127.0.0.1',
  // port: 3306,      // se puede cambiar el puerto de mysql

  //usuario base de datos (por defecto sería root, pero hay que evitarlo)
  user: 'ejemplouser',
  //contraseña del usuario en mysql
  password: '123456',
  //nombre del esquema al que nos vamos a conectar
  database: 'ejemplodb'
});

// Conexión a la base de datos
db.connect((err) => {
```

```
    if (err) {
      console.error('Error de conexión a la base de datos:', err);
    } else {
      console.log('Conexión exitosa a la base de datos');
    }
  });

// aquí van todas las rutas que queramos

// Ruta para obtener datos desde la base de datos
//cambiamos app por la nueva constante rutasCategorias
rutasCiudades.get('/', (req, res) => { //http://localhost:3000/ciudades
  // Consulta a la base de datos
  db.query('SELECT * FROM ciudades', (err, resultados) => {
    if (err) {
      console.error('Error al obtener datos desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json(resultados);
    }
  });
});

// Ruta para enviar datos y guardar en la base de datos
rutasCiudades.post('/enviar', (req, res) => { //http://localhost:3000/ciudades/enviar
  const { nombre, cantidad } = req.body;

  // Insertar datos en la base de datos
  db.query('INSERT INTO ciudades (nombre, cantidad) VALUES (?, ?)', [nombre,
cantidad], (err, resultado) => {
    if (err) {
      console.error('Error al guardar datos en la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json({ recibido: true, nombre, cantidad, id: resultado.insertId });
    }
  });
});

// Ruta para obtener un registro por ID
rutasCiudades.get('/:id', (req, res) => { //http://localhost:3000/ciudades/registro/3
  const idRegistro = req.params.id;

  // Consulta a la base de datos para obtener el registro por ID
  db.query('SELECT * FROM ciudades WHERE id = ?', [idRegistro], (err, resultados) => {
    if (err) {
      console.error('Error al obtener el registro desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      // Verifica si se encontró un registro
      if (resultados.length > 0) {
```

```
    res.json(resultados[0]); // Devuelve el primer resultado encontrado (debería ser único)
  } else {
    res.status(404).json({ error: 'Registro no encontrado' });
  }
}
});
});

//exportamos la constante routes
module.exports = rutasCiudades;
```

Sacamos la lógica a ficheros controladores:

En Express, es una buena práctica organizar tu código en controladores para manejar las distintas funciones o rutas de tu API. Aquí tienes un ejemplo básico de cómo podrían verse los controladores en una aplicación Express y Node.js.

Supongamos que tienes una entidad llamada "Usuario" y quieres realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en tu API. Aquí hay un ejemplo con controladores:

1. Estructura del proyecto:

plaintext

Copy code

```
- mi-api
  |- controllers
    |- usuariosController.js
  |- index.js
```


2. Crea el controlador para Usuarios (`usuariosController.js`):

javascript

Copy code

```
// controllers/usuariosController.js
const getAllUsuarios = (req, res) => {
  // Lógica para obtener todos los usuarios de la base de datos
  res.json({ mensaje: 'Obtener todos los usuarios' });
};

const getUserById = (req, res) => {
  const userId = req.params.id;
  // Lógica para obtener un usuario por ID de la base de datos
  res.json({ mensaje: `Obtener usuario por ID: ${userId}` });
};

const createUsuario = (req, res) => {
  const nuevoUsuario = req.body;
  // Lógica para crear un nuevo usuario en la base de datos
  res.json({ mensaje: 'Crear un nuevo usuario', usuario: nuevoUsuario });
};

const updateUsuario = (req, res) => {
```

```
const updateUsuario = (req, res) => {
  const userId = req.params.id;
  const datosActualizados = req.body;
  // Lógica para actualizar un usuario por ID en la base de datos
  res.json({ mensaje: `Actualizar usuario por ID: ${userId}`, datos: datosActualizados });
};

const deleteUsuario = (req, res) => {
  const userId = req.params.id;
  // Lógica para eliminar un usuario por ID de la base de datos
  res.json({ mensaje: `Eliminar usuario por ID: ${userId}` });
};

module.exports = {
  getAllUsuarios,
  getUserById,
  createUsuario,
  updateUsuario,
  deleteUsuario
};
```

3. Usa los controladores en tu aplicación (`index.js`):

```
javascript Copy code

// index.js
const express = require('express');
const bodyParser = require('body-parser');
const usuariosController = require('./controllers/usuariosController');

const app = express();
const port = 3000;

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(bodyParser.json());

// Rutas que utilizan los controladores
app.get('/api/usuarios', usuariosController.getAllUsuarios);
app.get('/api/usuarios/:id', usuariosController.getUsuarioById);
app.post('/api/usuarios', usuariosController.createUsuario);
app.put('/api/usuarios/:id', usuariosController.updateUsuario);
app.delete('/api/usuarios/:id', usuariosController.deleteUsuario);

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

En este ejemplo, cada función del controlador maneja una operación específica para la entidad "Usuario". Luego, en el archivo principal (`index.js`), se utilizan las funciones del controlador para manejar las rutas correspondientes de la API.

Esta estructura facilita la organización y mantenimiento del código, especialmente a medida que la aplicación crece en complejidad. Puedes expandir y ajustar según las necesidades específicas de tu aplicación.


Sacar la conexión a mysql en archivo aparte

En una estructura de API con archivos de controladores y archivos de rutas en Express, es preferible realizar la conexión a MySQL en un archivo separado, como por ejemplo en un módulo de base de datos. Esto ayuda a mantener una estructura más organizada y modular, lo que facilita el mantenimiento y la escalabilidad de tu aplicación.

Aquí hay un ejemplo de cómo podrías estructurar tu aplicación:

1. Estructura del proyecto:


plaintext

 Copy code

```
- mi-api
  |- controllers
  |   |- usuariosController.js
  |- routes
  |   |- usuariosRoutes.js
  |- database
  |   |- db.js
  |- index.js
```

2. Crea el archivo de conexión a la base de datos (`database/db.js`):

javascript

 Copy code

```
// database/db.js
const mysql = require('mysql');

const db = mysql.createConnection({
  host: 'localhost',
  user: 'tu_usuario_mysql',
  password: 'tu_contraseña_mysql',
  database: 'tu_base_de_datos'
});

db.connect((err) => {
  if (err) {
    console.error('Error de conexión a la base de datos:', err);
  } else {
    console.log('Conexión exitosa a la base de datos');
  }
});

module.exports = db;
```

3. Modifica el controlador y las rutas para utilizar la conexión a la base de datos

(`controllers/usuariosController.js` y `routes/usuariosRoutes.js`):

javascript

Copy code

```
// controllers/usuariosController.js
const db = require('../database/db');

const getAllUsuarios = (req, res) => {
  // Lógica para obtener todos los usuarios de la base de datos utilizando la conexión
  res.json({ mensaje: 'Obtener todos los usuarios' });
};

// Resto de las funciones del controlador

module.exports = {
  getAllUsuarios,
  // Resto de las funciones del controlador
};
```

javascript

Copy code

```
// routes/usuariosRoutes.js
const express = require('express');
const usuariosController = require('../controllers/usuariosController');

const router = express.Router();

// Rutas que utilizan los controladores
router.get('/', usuariosController.getAllUsuarios);
// Resto de las rutas

module.exports = router;
```

4. Usa los archivos de rutas en tu aplicación principal (`index.js`):

```
javascript Copy code

// index.js
const express = require('express');
const bodyParser = require('body-parser');
const usuariosRoutes = require('./routes/usuariosRoutes');

const app = express();
const port = 3000;

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(bodyParser.json());

// Rutas que utilizan los archivos de rutas
app.use('/api/usuarios', usuariosRoutes);
// Resto de las rutas

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

Con esta estructura, la conexión a la base de datos está encapsulada en un módulo separado (`database/db.js`), lo que facilita la gestión de la conexión y su reutilización en diferentes partes de la aplicación. Además, los controladores y las rutas pueden simplemente requerir este módulo para acceder a la conexión a la base de datos cuando sea necesario.

Servir archivos estáticos

Para que la ruta raíz (`/`) de tu aplicación Express sirva una página HTML, puedes utilizar el middleware `express.static` para servir archivos estáticos y enviar el archivo HTML como respuesta. A continuación, te mostraré un ejemplo sencillo:

1. Estructura del proyecto:

plaintext

Copy code

```
- mi-api
  |- public
    |- index.html
    |- index.js
```

En este ejemplo, el archivo `index.html` se encuentra en la carpeta `public` y contiene el contenido de tu página HTML.

2. Contenido de `index.html`:

html

Copy code

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Página de Inicio</title>
</head>
<body>
  <h1>Bienvenido a mi página de inicio</h1>
  <p>Esta es una página de ejemplo.</p>
</body>
</html>
```

3. Contenido de `index.js`:

```
javascript Copy code

// index.js
const express = require('express');
const path = require('path');

const app = express();
const port = 3000;

// Middleware para servir archivos estáticos desde la carpeta 'public'
app.use(express.static(path.join(__dirname, 'public')));

// Ruta raíz que sirve la página HTML
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

- `express.static`: Este middleware sirve archivos estáticos, como archivos HTML, CSS, JavaScript, etc., desde el directorio especificado (`public` en este caso). Se utiliza para servir los archivos estáticos de la página HTML.
- `app.get('/')`: Esta ruta maneja las solicitudes GET a la ruta raíz (`/`). Cuando un usuario accede a la ruta raíz, se envía el archivo HTML como respuesta utilizando `res.sendFile`.
- `path.join(__dirname, 'public', 'index.html')`: Se utiliza para construir la ruta completa al archivo HTML. `__dirname` representa el directorio actual del archivo en el que se está ejecutando el código.

Ahora, cuando ejecutas tu aplicación y accedes a `http://localhost:3000/`, deberías ver la página HTML en la ruta raíz. Ajusta los nombres de archivos y carpetas según la estructura específica de tu proyecto.

CORS y Fetch desde otro equipo

Cuando estás haciendo una solicitud `fetch` a una API en otro equipo (dominio), es posible que te encuentres con problemas de política de mismo origen (Same-Origin Policy). La política de mismo origen es una política de seguridad en los navegadores web que restringe cómo los scripts en una página web pueden hacer solicitudes a otra página.

Para permitir solicitudes entre dominios diferentes, debes configurar el servidor para incluir encabezados CORS (Cross-Origin Resource Sharing). Estos encabezados le dicen al navegador que es seguro permitir solicitudes desde un dominio específico.

A continuación, te mostraré cómo configurar CORS en un servidor Express y cómo hacer una solicitud `fetch` desde el lado del cliente:

Configuración de CORS en Express

Instala el paquete `cors` en tu proyecto:

```
bash Copy code  
  
npm install cors
```


Luego, en tu servidor Express (`index.js`), agrega la configuración de CORS:

```
javascript Copy code  
  
// index.js  
const express = require('express');  
const cors = require('cors');  
const path = require('path');  
  
const app = express();  
const port = 3000;  
  
// Configurar CORS  
app.use(cors());  
  
// Resto del código...
```


Lado del Cliente (HTML y JavaScript)

En tu archivo HTML (`public/index.html`), puedes tener un botón o algún evento que desencadene la solicitud `fetch`:

html

 Copy code

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Solicitud Fetch a API Externa</title>
</head>
<body>
  <button id="btnFetch">Hacer solicitud Fetch</button>

  <script src="js/hacerFetch.js"></script>
</body>
</html>
```

En tu archivo JavaScript (`public/js/hacerFetch.js`), puedes hacer la solicitud `fetch`:

```
javascript Copy code

// public/js/hacerFetch.js

document.getElementById('btnFetch').addEventListener('click', () => {
  // Reemplaza la URL con la URL de la API externa que deseas acceder
  const apiUrl = 'https://api.externa.com/data';

  fetch(apiUrl)
    .then(response => {
      if (!response.ok) {
        throw new Error(`Error de red - ${response.status}`);
      }
      return response.json();
    })
    .then(data => {
      console.log('Datos recibidos:', data);
      // Haz algo con los datos recibidos
    })
    .catch(error => console.error('Error en la solicitud Fetch:', error));
});
```

Asegúrate de reemplazar `https://api.externa.com/data` con la URL real de la API externa a la que estás intentando acceder.

Ten en cuenta que algunos servidores también pueden requerir autenticación, como claves API, tokens de acceso, etc. Asegúrate de seguir la documentación de la API externa para obtener detalles sobre cómo autenticarte correctamente si es necesario.

No es necesario abrir el Puerto de express en firewall



You

por qué no es necesario abrir el puerto express para que se consuma la api desde otro equipo?



ChatGPT

Cuando creas una API con Express y la expones en un servidor, por defecto, Express escucha en un puerto específico en la máquina donde se está ejecutando. Este puerto es donde la aplicación Express acepta solicitudes HTTP entrantes. Sin embargo, para que la API sea accesible desde otros equipos, no es necesario "abrir el puerto" en el sentido de abrir puertos en un firewall. La accesibilidad desde otros equipos depende de varios factores:

1. Dirección IP y Puerto:

- Si tu servidor Express está escuchando en una dirección IP específica (por ejemplo, `127.0.0.1` para conexiones solo locales) o en todas las interfaces (`0.0.0.0`), afectará si la API es accesible desde otros equipos.
- Si estás ejecutando Express en `localhost` o `127.0.0.1`, la API solo será accesible localmente. Para que sea accesible desde otros equipos, debes configurar Express para escuchar en la dirección IP de tu máquina o en `0.0.0.0` (todas las interfaces).

2. Firewall del Servidor:

- Asegúrate de que el firewall en el servidor permita conexiones al puerto en el que Express está escuchando. Si el puerto está bloqueado, las solicitudes no podrán llegar a tu aplicación.

3. Firewall del Router:

- Si estás detrás de un router, asegúrate de que el puerto en el que Express está escuchando esté redirigido correctamente desde el router al servidor. Esto se llama reenvío de puertos y es esencial para permitir que las solicitudes externas lleguen al servidor.

4. Configuración CORS:

- En el lado del servidor (Express), asegúrate de que has configurado CORS adecuadamente para permitir solicitudes desde dominios específicos o cualquier origen (`*`). Esto es necesario para evitar problemas de política de mismo origen en los navegadores web.

En resumen, para que una API Express sea accesible desde otros equipos, debes asegurarte de que la aplicación esté escuchando en la dirección IP correcta, que el puerto esté abierto en el firewall del servidor y, si es necesario, configurar el reenvío de puertos en el router. También, asegúrate de que CORS esté configurado adecuadamente para permitir solicitudes desde los dominios deseados.

Actuaciones con el puerto de MySql



You

¿por qué no hay que abrir el puerto de mysql si está en el mismo servidor que express?



ChatGPT

Cuando tu aplicación Express y tu base de datos MySQL están en el mismo servidor, no necesitas "abrir el puerto" de MySQL para acceder desde la aplicación Express en el mismo servidor. Esto se debe a que las conexiones locales a la base de datos generalmente no pasan por el mismo proceso de enrutamiento que las conexiones externas.



Cuando tu aplicación Express y tu base de datos MySQL están en el mismo servidor, pueden comunicarse a través de conexiones locales (localhost o 127.0.0.1). En este caso, la comunicación se produce a través de un socket Unix o una conexión de bucle de retroalimentación, y no pasa por la red física. Por lo tanto, no es necesario abrir un puerto específico para MySQL en el firewall del servidor.

Si tu aplicación Express y tu base de datos MySQL están en servidores separados, entonces necesitarás configurar MySQL para permitir conexiones remotas y abrir el puerto correspondiente en el firewall del servidor MySQL para que las conexiones externas sean posibles.