RETO DOS:

DEPARTAMENTO DE INFORMÁTICA
NIEVES RUIZ NOGUERAS

Curso 2025-2026

Capítulo 1

Node.js y npm

Figura 1.1: Node.js

Figura 1.2: npm

1.1. Introducción

https://nodejs.org/es/

https://www.youtube.com/watch?v=xJzzu7MVZXw (2:28)

Node.js es un entorno de ejecución para javascript construido con el motor de javascript V8 de Google Chrome. Hasta ahora sólo podíamos ejecutar javascript desde un navegador, con node.js podemos ejecutar js en el servidor.

Una definición más completa, extraída de la wikipedia, es la siguiente: **Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.**

Para que javascript sea un lenguaje servidor, las características que han sido agregadas en node.js son:

- Organización del código con piezas reutilizables (módulos).
- Manejo de archivos (filesystem).
- Manejo de bases de datos.
- Interacción con web services.
- Interacción mediante el protocolo HTTP.
- Manejo de procesos que puedan tardar mucho tiempo en ser ejecutados.

npm (Node Package Manager) es el sistema gestor de paquetes de node. Es un gestor de paquetes desarrollado en su totalidad bajo el lenguaje JavaScript por Isaac Schlueter, a través del cual podemos obtener cualquier librería con tan solo una línea de código, lo que nos permitirá agregar dependencias de forma simple, distribuir paquetes y administrar eficazmente tanto los módulos como el proyecto a desarrollar en general.

npm está formado de dos partes principales. Un **repositorio** online para publicar paquetes de software libre para ser utilizados en proyectos Node.js y una herramienta para la **terminal** (command line utility) para interactuar con dicho repositorio que te ayuda a la instalación de utilidades, manejo de dependencias y la publicación de paquetes.

1.2. Empezar a trabajar con node.js

Desde la página oficial https://nodejs.org/es/ podemos descargar la última versión de node.js.

Tras instalarlo en nuestro equipo y desde Visual Studio, podemos empezar a crear nuestros primeros ficheros js para ser ejecutados en el servidor.

En la siguiente imagen podemos ver un primer ejemplo y el resultado de su ejecución, tras comprobar la versión de node.js instalada en el equipo.

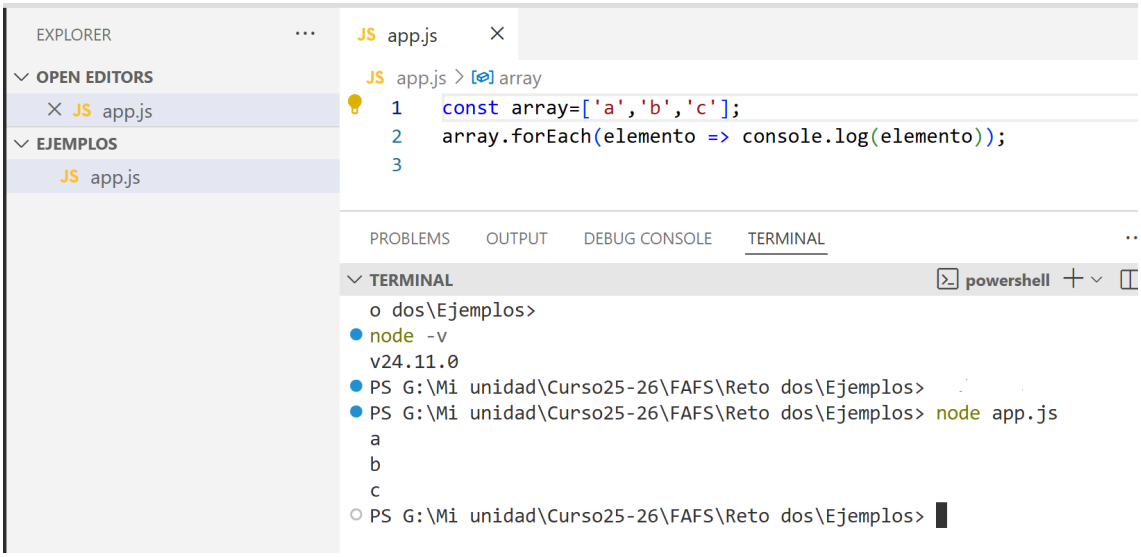


Figura 1.3: app.js

Destacar que no se abre el navegador. No estamos ejecutando en el entorno cliente.

Node.js tiene un **sistema de módulos** incorporado. Un archivo node.js puede importar la funcionalidad expuesta por otro archivo node.js.

En node.js, un módulo es un conjunto de funciones y objetos de JavaScript que las aplicaciones externas pueden usar.

Debido a que los módulos proporcionan funciones que pueden reutilizarse en muchos programas, permiten crear aplicaciones acopladas libremente que se adaptan a la complejidad.

Este sistema de módulos se basa en importar y en exportar. El sistema clásico de módulos (commonJS (**CJS**)) usa **export** para exportar y **require** para importar). El sistema de módulos ES Modules (**ESM**) es el standard oficial y usa **import** y **export**. Este último es el que se usa en proyectos modernos.

A través del fichero package.json, se le indica a node que sistema utilizar.

```
1
2  // CJS
3
4  {
5    "name": "ejercicioCuatro",
6    "version": "1.0.0",
7    "type": "module",
8    "dependencies": {
9      "chalk": "^5.3.0"
10   }
11 }
```

Ejemplo exportando una función:

```
1  Fichero saludar.js
2
3  var saludar = function(){
4      console.log('Hola');
5  }
6
7  module.exports = saludar;
```

```
1  Fichero appsaludar.js
2
3  var saludar = require('./saludar')
4  saludar();
```

El mismo ejemplo con modules:

```
1
2  export default function saludar() {
3      console.log("Hola");
4  }
```

```

import saludar from "./saludar.js";
saludar();
```

Ejemplo exportando un dato (el array frutas):

```

// frutas.js (CommonJS)

const frutas = ["manzana", "pera", "plátano", "kiwi"];

module.exports = frutas;
```

```
// appFrutas.js (CommonJS)

const frutas = require("./frutas");

console.log("Listado de frutas:");
console.log(frutas);
```

```

// frutas.js (ES Modules)

const frutas = ["manzana", "pera", "plátano", "kiwi"];

export default frutas;
```

```

// appFrutas.js (ES Modules)

import frutas from "./frutas.js";

console.log("Listado de frutas:");
console.log(frutas);
```

1.3. El manejador de paquetes npm

Un manejador de paquetes es un software que permite la instalación y actualización de los paquetes. Los paquetes contienen código que puede ser utilizado y por ende distribuido a través de un manejador de paquetes. Un paquete puede funcionar de forma aislada pero también puede definir dependencias, es decir que depende de otros paquetes para funcionar. Los paquetes se actualizan periódicamente.

La lista de paquetes que podemos utilizar se puede encontrar en npmjs.com. Debido a que cualquier persona puede crear un paquete, es importante tener en cuenta a la hora de utilizarlo el grado de confianza en el desarrollador, la

licencia que utiliza, cuantas personas lo están utilizando y el mantenimiento que se le da a dicho paquete.

Algunos de los paquetes más populares de Node.js son los siguientes:

- **Express**: es un marco de aplicaciones web para Node.js que proporciona un conjunto de características para construir aplicaciones web y APIs.

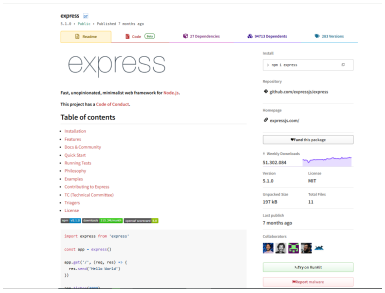


Figura 1.4: Paquete express

- **Socket.IO**: es una biblioteca que permite la comunicación en tiempo real entre clientes y servidores.
- **MongoDB**: es un sistema de base de datos no relacional que se puede usar con Node.js.
- **Passport**: es un marco de autenticación para Node.js que permite a los desarrolladores implementar fácilmente autenticación de usuarios en sus aplicaciones.
- **AWS SDK**: es un conjunto de herramientas de desarrollo de software para el servicio de nube de Amazon Web Services (AWS).
- **Nodemon**: es una herramienta que permite a los desarrolladores reiniciar automáticamente sus aplicaciones de Node.js cuando se detectan cambios en el código fuente.
- **async**: es una biblioteca que proporciona funciones asíncronas para Node.js, lo que permite a los desarrolladores escribir código asíncrono de manera más sencilla.
- .....

Para instalar un paquete, debemos ir a la linea de comandos y teclear **npm install o npm i**.

```
npm install nombre_del_paquete

PS C:\Users\Egibide\Desktop\EjemplosNodejs> npm install typescript
Debugger attached.

changed 1 package, and audited 2 packages in 20s

found 0 vulnerabilities
Waiting for the debugger to disconnect...
PS C:\Users\Egibide\Desktop\EjemplosNodejs>
```

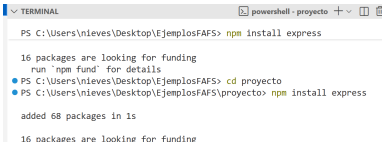


Figura 1.5: Paquete express instalado

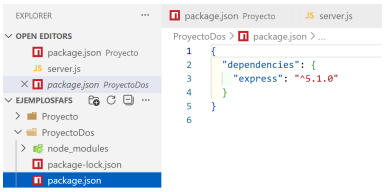


Figura 1.6: Paquete express instalado

Esta orden instala un paquete localmente, lo que significa que sólo se puede utilizar la funcionalidad del paquete en el directorio actual. Los paquetes se pueden instalar de forma global utilizando el parámetro (**-g**) para que se pueda acceder a él desde cualquier directorio del ordenador.

El software npm descargará el paquete de manera automática junto con sus dependencias y realizará la instalación.

Al instalar un paquete aparece la **carpeta node-modules**. Es un directorio que se crea en la carpeta raíz de nuestro proyecto cuando instalamos en local paquetes o dependencias mediante npm. De esta forma, desde nuestro código Javascript podemos importar paquetes externos instalados mediante npm, teniéndolos en nuestro proyecto local (no dependen de una ruta externa al proyecto) y sin necesidad de manipularlos manualmente.

El fichero **package.json** contiene un información sobre el proyecto (nombre,

versión, etc...) además de listar los paquetes de los que depende.. Se crea ejecutando **npm init**.

```
PS C:\Users\nieves\Desktop\EjemplosFAFS\proyecto> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
    defaults.

See ‘npm help init‘ for definitive documentation on these fields
and exactly what they do.

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

Press ^C at any time to quit.

package name: (proyecto) proyectouno
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to
    C:\Users\nieves\Desktop\EjemplosFAFS\proyecto\package.json:

{
  "dependencies": {
    "express": "^5.1.0"
  },
  "name": "proyectouno",
  "version": "1.0.0",
  "main": "index.js",
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": ""
}


Is this OK? (yes)
PS C:\Users\nieves\Desktop\EjemplosFAFS\proyecto>
```

Contenido del fichero:

```
{
  "dependencies": {
    "express": "^5.1.0"
  },
  "name": "proyectouno",
  "version": "1.0.0",
  "main": "index.js",
  "devDependencies": {},
```

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Para saltarte el proceso de proporcionar la información tú mismo, puedes simplemente ejecutar el comando **npm init -y**

Tras la instalación un paquete de forma local, el contenido del fichero ha variado y también ha aparecido un nuevo fichero.

El fichero **package-lock.json** es un archivo generado por npm que proporciona una representación detallada de todas las dependencias de un proyecto y de sus versiones específicas. Esto es útil porque asegura que cualquier persona que clone el proyecto y ejecute npm install obtendrá exactamente las mismas versiones de paquetes.

Además, el archivo package-lock.json también puede ayudar a resolver problemas de dependencias que puedan surgir al trabajar en un proyecto en equipo, ya que asegura que todos los desarrolladores estén utilizando las mismas versiones de paquetes.

Las principales propiedades del fichero package.json:

https://medium.com/noders/t%C3%BA-yo-y-package-json-9553929fb2e3

- **name** Nombre del proyecto, librería o paquete. Se recomienda que coincida con el repositorio.

```
"name": "ejemplosnodejs"
```

- **version number**

```
"version": "1.0.0",
```

- Indica el archivo principal o de entrada a la aplicación.

```
"main": "app.js",
```

- **licenses** Tipo de licencia del proyecto o paquete. Por defecto ISC (Licencia de software libre permisiva).

  https://spdx.org/licenses/

- **dependencies** Colección de paquetes para producción y la versión instalada.

```
1  "dependencies": {
2     "typescript": "^4.9.4"
3   }
```

- **scripts** (comandos) npm crea un apartado de scripts donde solo tenemos un script llamado test. Este script, por defecto no realiza ninguna tarea, simplemente es una llamada a echo mostrando que no existen tests.

```
1  "scripts": {
2     "test": "echo \"Error: no test specified\" && exit 1"
3   },
```

Este script se puede modificar (o borrar) con seguridad, ya que no hace nada. L o interesante es crear nuevos scripts con tareas que realizamos frecuentemente.

Tarea	Descripción	Comando
start	Se suele usar para tareas de inicio del proyecto. Se puede arrancar con npm start.	npm run start
dev	Alternativa al anterior, se suele usar para levantar servidores de desarrollo locales.	npm run dev
serve	Idem al anterior.	npm run serve
build	Tarea que construye los ficheros finales para subir a la web de producción.	npm run build
test	Suele iniciar una batería de tests. Se puede arrancar con npm test o npm t.	npm run test
deploy	Suele desplegar en la web de producción la webapp construída con build.	npm run deploy

Figura 1.7: Scripts

Estos scripts se pueden usar para automatizar tareas como compilar, probar e implementar el código. Los scripts se pueden ejecutar desde la línea de comandos mediante el npm run comando.

1.3.1. Más tareas que se pueden realizar con npm

npm start sirve para iniciar una aplicación . Ejecuta los comando definidos en la sección scripts.start del archivo package.json

Figura 1.8: npm

npm install package-name Instalar un paquete. Instala la última versión estable del paquete nombrado.

npm install pakage-name@version Igual que el anterior, con la diferencia que el parámetro version nos permite indicar la versión exacta del módulo que vamos a instalar

npm install -g pakage-name El parámetro -g instala el módulo de forma global, lo que hace disponible en cualquier desde cualquier ubicación o proyecto.

npm uninstall package-name Desinstalar un módulo.

npm uninstall -g package-name Igual que el anterior, con la diferencia que el parámetro -g le indica que borre la dependencia de las librerías globales.

npm list -g -depth 0 Listar módulos globales instalados. El parámetro list es para listar todos los módulos instalados en la carpeta actual, -g complementa al comando anterior, para indicar que solo muestre los paquetes globales instalados en la carpeta del usuario.

El parámetro –depth 0 filtra las dependencias de cada paquete en la vista de árbol.

npm update Actualiza todos los paquetes del package.json.

npm update package-name Actualiza un paquete.

npm outdated Imprime una lista con el nombre de los paquetes desactualizados.

npm run Ejecutar comandos o scripts.

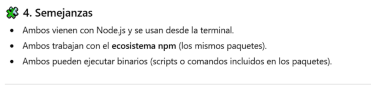
1.4. npx

npx es una herramienta que se incluye en la instalación de Node.js. Permite ejecutar paquetes de npm **sin tener que instalarlos** de manera global en el sistema. También proporciona una manera de ejecutar una versión específica de un paquete sin tener que instalar ninguna versión en particular de manera permanente. Esto puede ser útil para probar nuevas versiones de un paquete o para ejecutar un paquete temporalmente en un proyecto.

🧠 3. Diferencias clave

Característica	`npm`	`npx`
Función principal	Instalar y gestionar paquetes	Ejecutar paquetes (sin necesidad de instalación previa)
Instala paquetes	Sí	No necesariamente
Viene con Node.js	Sí	Sí (desde npm 5.2+)
Uso típico	`npm install react`	`npx create-react-app mi-app`
Necesidad de instalación previa	Sí (el paquete debe estar instalado)	No (puede descargarlo y usarlo directamente)

Figura 1.9: Diferencias npm y npx



🧩 4. Semejanzas

- Ambos vienen con Node.js y se usan desde la terminal.
- Ambos trabajan con el **ecosistema npm** (los mismos paquetes).
- Ambos pueden ejecutar binarios (scripts o comandos incluidos en los paquetes).

Figura 1.10: Semejanzas npm y npx

🚀 Ejemplo comparativo

Con `npm`:

```bash
npm install create-react-app -g
create-react-app mi-app
```

Con `npx`:

```bash
npx create-react-app mi-app
```

👉 Resultado igual, pero con `npx` no llenas tu sistema de instalaciones globales innecesarias.

Figura 1.11: Ejemplo comparativo npm y npx

1.5. nodemon

nodemon (Node Monitor) es una herramienta que **reinicia automáticamente tu aplicación Node.js cada vez que detecta un cambio** en los archivos del proyecto.

En lugar de tener que parar y volver a ejecutar manualmente node app.js cada vez que editas el código, nodemon lo hace automáticamente por ti.

```
1
2 // Instalación global lo que permite usar nodemon desde cualquier proyecto
3 npm install -g nodemon
4
5 nodemon app.js
6
7 // Instalación local (recomendada para proyectos). Está línea agrega
      nodemon como una dependencia en el fichero package.json
8
9 npm install --save-dev nodemon
```

A la hora de ejecutar nuestra aplicación, usaremos nodemon en vez de node y cada vez que se cambie algo en app.js o en cualquier otro archivo, nodemon reiniciará el servidor automáticamente.

```
1
2 nodemon app.js
```

1.6. Servidor HTTP

Como ya sabemos Hypertext Transfer Protocol o protocolo de transferencia de hipertexto es el protocolo de comunicación que permite las transferencias de información en la World Wide Web. Intercambio de información entre cliente y servidor. El servidor queda a la espera de alguna solicitud HTTP ejecutada por el cliente y proporciona una respuesta.

Node.js viene con algunos módulos que no necesitan instalación, uno de ellos es el http.

```
1
2  const http = require('http');
3  const url = require('url');
4
5  const server = http.createServer((req, res) => {
6    // Parseamos la URL para obtener los parámetros
7    const parsedUrl = url.parse(req.url, true);
8    const name = parsedUrl.query.name || 'desconocido';
9
10   // Configuramos la cabecera y la respuesta
11   res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
12   res.end(`Hola, ${name}!`);
13 });
14
15 const PORT = 3000;
16 server.listen(PORT, () => {
17   console.log(`Servidor HTTP escuchando en http://localhost:${PORT}`);
18 });
```

Crear un servidor http en node.js es fácil. Con **require** hay que importar el módulo http (no necesita instalación, viene ya por defecto). Con **createserver** se crea el servidor y se define la respuesta, se indica el **puerto** a través del cual comunicarse y se pone a la **escucha**. Cuando desde un navegador se haga la petición (`http://localhost:3000/?name=Maria`) a través del puerto indicado, es decir, cuando haya un requerimiento, se responderá.

url.parse parsea, es decir, convierte la URL en un objeto más fácil de usar. El parámetro true le dice que convierta los parámetros de la query (?name=Maria) a un objeto JavaScript. parsedUrl.query.name proporciona el valor.

1.7. Express

```
npm install express
```

El paquete express es un framework de aplicación de Node.js para **crear aplicaciones web y APIs**. Es muy popular debido a su facilidad de uso y a la gran cantidad de características que ofrece. Algunas de las cosas que puedes hacer con express incluyen:

- Configurar **rutas** y manejar solicitudes HTTP.
- Proporcionar **middleware** para personalizar el comportamiento de la aplicación.

  Middleware es una función que se ejecuta en medio de la cadena de procesamiento de una solicitud HTTP. Se puede utilizar el middleware para realizar una serie de tareas, como:

  - Analizar la solicitud y el cuerpo de la solicitud.
  - Autenticar al usuario.
  - Generar una respuesta.
  - Modificar el objeto de solicitud o de respuesta.

  Express tiene una gran cantidad de middleware disponibles de forma predeterminada, y también nos permite crear los nuestros.
- Utilizar **plantillas** para generar contenido dinámicamente.
- Integrarse con **bases de datos** y realizar operaciones CRUD.

Tras instalar este paquete (npm i express) la creación de un servidor http se realiza de la siguiente manera:

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

// Middleware para leer JSON en peticiones POST
app.use(express.json());

// Ruta GET: recibe el nombre por query
app.get('/saludo', (req, res) => {
  const name = req.query.name || 'desconocido';
  res.send(`Hola, ${name}!`);
});

```

```
14 // Ruta POST: recibe el nombre en el cuerpo JSON
15 app.post('/saludo', (req, res) => {
16   const { name } = req.body;
17   res.send(`Hola, ${name || 'desconocido'}!`);
18 });
19
20 const PORT = 3000;
21 app.listen(PORT, () => {
22   console.log(`Servidor Express escuchando en http://localhost:${PORT}`);
23 });
24
25
26 // Probarlo con http://localhost:3000/saludo?name=Carlos
```

Comparación rápida

Característica	HTTP nativo	Express
Verbos HTTP	Manual	Simplificados (`app.get`, `app.post`, etc.)
Manejo de JSON	Hay que hacerlo manualmente	Automático con `express.json()`
Código	Más largo y verboso	Más limpio y legible
Recomendado para	Aprender lo básico	Proyectos reales

Figura 1.12: Http vs express

```
1
2  // Si estás en Node 18+ ya viene fetch incluido
3  // Si no, instala con: npm install node-fetch
4
5  // Ejemplo de función para probar ambos servidores
6  async function probarServidores() {
7    const nombre = 'Lucía';
8
9    // --- GET al servidor HTTP ---
10   const resHttp = await fetch(`http://localhost:3000/?name=${nombre}`);
11   const textoHttp = await resHttp.text();
12   console.log('Respuesta del servidor HTTP:', textoHttp);
13
14   // --- GET al servidor Express ---
15   const resExpress = await
         fetch(`http://localhost:4000/saludo?name=${nombre}`);
16   const textoExpress = await resExpress.text();
17   console.log('Respuesta del servidor Express (GET):', textoExpress);
18
19   // --- POST al servidor Express ---
20   const resPost = await fetch('http://localhost:4000/saludo', {
21     method: 'POST',
22     headers: { 'Content-Type': 'application/json' },
23     body: JSON.stringify({ name: nombre }),
24   });
25   const textoPost = await resPost.text();
26   console.log('Respuesta del servidor Express (POST):', textoPost);
27 }
28
29 probarServidores().catch(console.error);
```

1.8. MySQL en un proyecto Node.js con Express

Node.js permite comunicarse con MySQL mediante drivers como **mysql2**.

```
1
2 npm install mysql2
```

Al combinarlo con Express se obtiene una API (Interacción de una Aplicación Web) (Application Programming Interface)) estructurada, peticiones rápidas y una integración completa con frontends mediante JSON.

La conexión se debe crear en un archivo llamado probablemente **db.js**.

```
1
2  const mysql = require("mysql2");
3
4  const db = mysql.createConnection({
5      host: "localhost",
6      user: "root",
7      password: "usbw",
8      database: "bdxxxxx",
9      port: 3307
10 });
11
12 db.connect(err => {
13     if (err) console.error("Error MySQL:", err);
14     else console.log("Conectado a MySQL");
15 });
16
17 module.exports = db;
18
19 /*
20
21 host: dirección del servidor MySQL (normalmente localhost)
22
23 user: usuario de MySQL (en USBWebserver normalmente root)
24
25 password: contraseña del usuario (en USBWebserver normalmente usbw)
26
27 database: la base de datos con la que trabajar
28
29 port: puerto MySQL (3306 o 3307 por defecto)
```

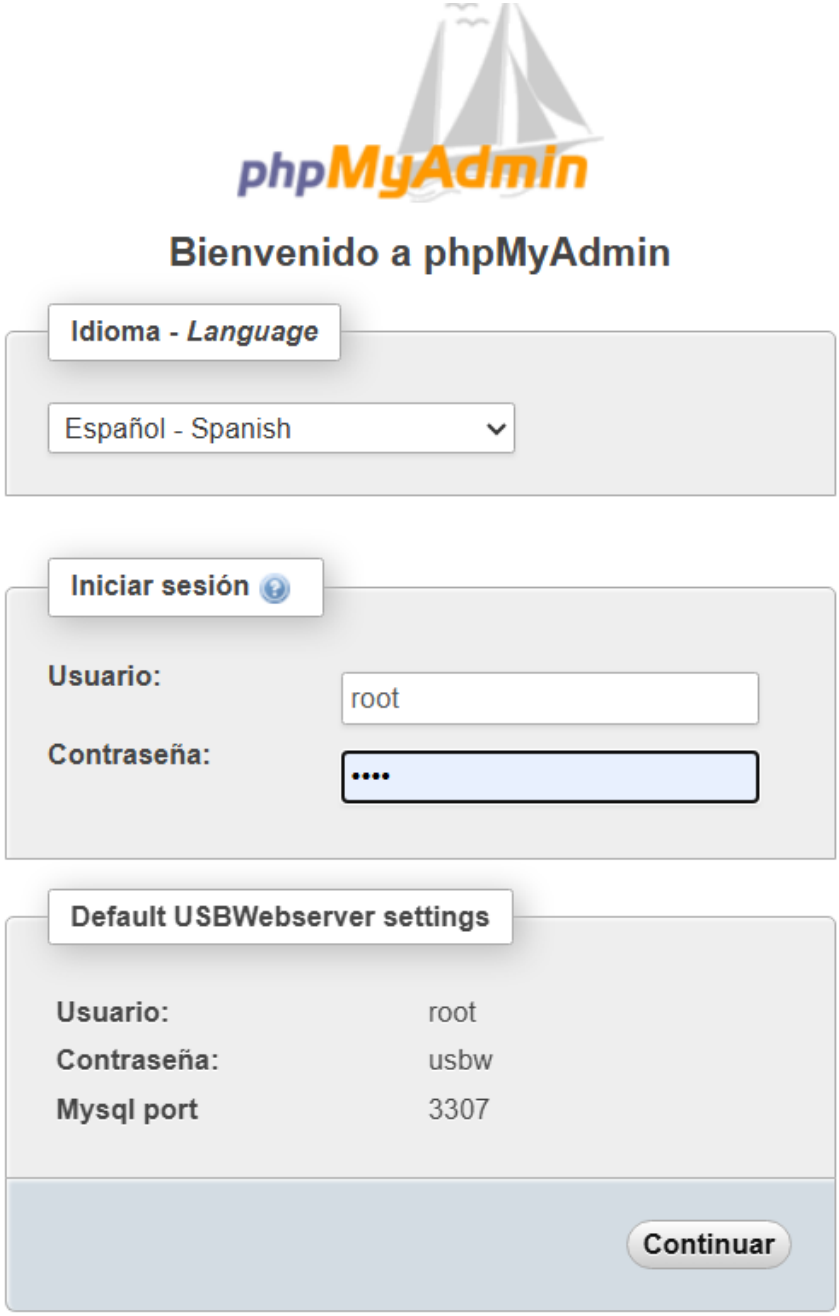


Figura 1.13: MySql dentro de UsbWebServer

La conexión **se exporta** para que otros archivos del proyecto la puedan utilizar.

En el patrón MVC (Modelo Vista Controldor), los **modelos** son los encargados de comunicarse con la base de datos. Estos archivos contienen solo SQL, sin lógica del servidor ni cosas de Express.

```
1
2  // modelo/xxxxModel.js
3
4  const db = require("../db");
5
6  module.exports = {
7      getxxx(callback) {
8          db.query("SELECT * FROM ....", callback);
9      },
10
11     createxxx(nombre, callback) {
12         db.query("INSERT INTO ...., callback);
13     },
14
15     updatexxx(id, nombre, callback) {
16         db.query("UPDATE ...., callback);
17     },
18
19     deletexxx(id, callback) {
20         db.query("DELETE FROM ...", [id], callback);
21     }
22 };
```

Añadiendo los controladores, la estructura del proyecto queda como la mostrada en la siguiente imagen:

:

Estructura final (Express + MySQL + Modelo + Controladores)

```
project/
 ├── server.js
 ├── db.js
 ├── models/
 │     └── userModel.js
 ├── controllers/
 │     └── userController.js
 └── public/
       ├── index.html
       ├── app.js
       └── styles.css
```

Figura 1.14: Estructura MVC

Ejemplo de controlador

```
1
2  const User = require("../models/userModel");
3
4  module.exports = {
5      getUsers(req, res) {
6          User.getUsers((err, results) => {
7              if (err) return res.status(500).json({ error: err });
8              res.json(results);
9          });
10     },
```

Ejemplo de server

```
1
2  const express = require("express");
3  const app = express();
4  const userController = require("./controllers/userController");
5
6  app.use(express.json());
7  app.use(express.static("public"));
8
9  app.get("/api/users", userController.getUsers);
10
11
12 app.listen(3000, () => {
13     console.log("Servidor en http://localhost:3000");
14 });
```