Express + Java Script

Ing. David Lozada

Express

Express (o **Express.js**) es un **framework web para Node.js** que te permite crear aplicaciones web y APIs de forma rápida, sencilla y eficiente.

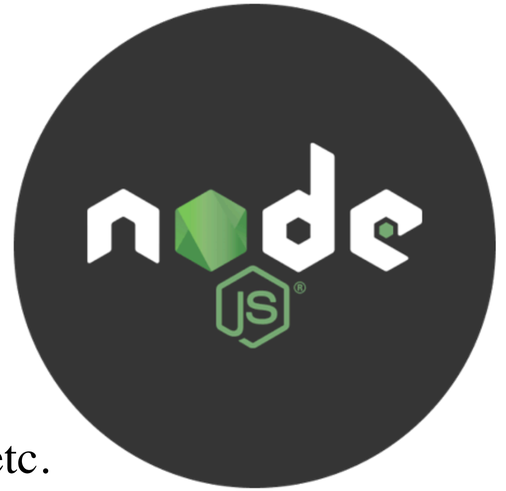
¿Qué significa esto?

- **Node.js** permite ejecutar JavaScript en el servidor.
- **Express** es una *capa adicional* que facilita muchas tareas comunes en el desarrollo backend.

¿Para qué se usa Express?

Con Express puedes:

- Crear servidores web.
- Construir APIs REST (GET, POST, PUT, DELETE).
- Gestionar rutas y peticiones HTTP.
- Integrar bases de datos como MongoDB, MySQL, PostgreSQL, etc.
- Renderizar vistas (HTML) con motores como EJS o Pug.



Express

¿Por qué usar Express?

Ventaja	Explicación
🟢 Simple y minimalista	No impone una estructura rígida, tú decides cómo organizar.
⚡ Rápido	Basado en Node.js, altamente eficiente.
🔌 Extensible	Puedes agregar middlewares fácilmente.
📚 Comunidad amplia	Mucha documentación, tutoriales y módulos disponibles.

En resumen:

Express es una herramienta que hace más fácil crear servidores y APIs en Node.js.

Es ideal para desarrollar desde aplicaciones pequeñas hasta grandes sistemas backend.

¿Quieres que te muestre cómo crear una API básica con Express?

Instalación Express

1. Requisitos previos

Antes de comenzar con Express, necesitas:

- Conocer **JavaScript (ES6)** básico.
- Tener instalado **Node.js** y **npm**.

Puedes instalar Node.js desde https://nodejs.org

```
mkdir mi-app-express
cd mi-app-express
npm init -y
```

2. Crear tu primer proyecto con Express

Paso 1: Inicializa el proyecto

Paso 2: Instala Express

npm install express

Paso 3: Crea un archivo `index.js`

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

app.get('/', (req, res) => {
  res.send('¡Hola mundo desde Express!');
});

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

Instalación Express

1. Requisitos previos

Antes de comenzar con Express, necesitas:

- Conocer **JavaScript (ES6)** básico.
- Tener instalado **Node.js** y **npm**.

Puedes instalar Node.js desde https://nodejs.org

```
mkdir mi-app-express
cd mi-app-express
npm init -y
```

2. Crear tu primer proyecto con Express

Paso 1: Inicializa el proyecto

Paso 2: Instala Express

npm install express

Paso 3: Crea un archivo `index.js`

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

app.get('/', (req, res) => {
  res.send('¡Hola mundo desde Express!');
});

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

Instalación Express

Paso 4: Ejecuta tu servidor

```
node index.js
```

Abre tu navegador y visita `http://localhost:3000`

3. Temas clave que debes aprender

Rutas

```
app.get('/usuarios', (req, res) => { ... });
app.post('/usuarios', (req, res) => { ... });
```

Middleware

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

Instalación Express

Parámetros en rutas

```
app.get('/usuario/:id', (req, res) => {
  const id = req.params.id;
});
```

Enrutadores (Routers)

Para organizar mejor las rutas en varios archivos.

Controladores y modelos (estructura MVC básica)

4. Proyectos para practicar

- API REST de tareas (CRUD)
- API para manejar productos o usuarios
- Backend para una app de notas
- Servidor para una app con React frontend

Instalación Express

5. Herramientas útiles

- Postman o Insomnia para probar tu API
- Nodemon (reinicia el servidor automáticamente):

```
npm install --save-dev nodemon
```

6. Recursos recomendados

- Documentación oficial: https://expressjs.com/es/
- Curso gratuito en YouTube: busca “Curso Express JS desde cero”
- Libros: “Node.js y Express” de Ethan Brown

Ejemplo Basico


```
// Cargar el módulo express
const express = require('express');

// Crear una aplicación express
const app = express();

// Definir el puerto
const PORT = 3000;

// Ruta de ejemplo
app.get('/', (req, res) => {
  res.send('¡Hola mundo desde Express!');
});

// Iniciar el servidor
app.listen(PORT, () => {
  console.log(`Servidor funcionando en http://localhost:${PORT}`);
});
```

Ejemplo Basico

Ejecuta tu servidor

```
node index.js
```

Abre tu navegador en http://localhost:3000

3. Aprender los conceptos básicos

Rutas básicas

```
app.get('/saludo', (req, res) => {
  res.send('Hola desde /saludo');
});
```

Las **rutas en Express.js** se utilizan para **definir cómo debe responder tu servidor** a diferentes **peticiones HTTP** (como GET, POST, PUT, DELETE) que llegan a determinadas **URLs**.
Son uno de los pilares principales del desarrollo backend.

🛣️ ¿Qué es una ruta?

Una **ruta** es como una "dirección" que el cliente (por ejemplo, un navegador o Postman) usa para pedir algo al servidor. En Express, cada ruta se define con una función que se ejecuta cuando llega una petición a una URL específica.

Arreglos Json

Un **arreglo JSON** es una estructura de datos en formato **JSON (JavaScript Object Notation)** que permite almacenar una colección ordenada de elementos.

Se desglosará en tres partes:

1. Arreglos JSON

Un **arreglo (array)** en JSON es una lista de valores encerrada entre corchetes `[ ]`.
Cada valor puede ser:

- Un número
- Una cadena (string)
- Un booleano (`true` o `false`)
- `null`
- Un objeto `{ }`
- Otro arreglo `[ ]`

Ejemplo de arreglo JSON simple:

```
[
  "Colombia",
  "México",
  "Argentina"
]
```

Arreglos Json

Ejecuta tu servidor

```
node index.js
```

Abre tu navegador en http://localhost:3000

3. Aprender los conceptos básicos

Enviar datos JSON

```
app.get('/datos', (req, res) => {
  res.json({ nombre: 'David', edad: 41 });
});
```

Un **JSON** (JavaScript Object Notation) es un **formato ligero de intercambio de datos**, muy utilizado en aplicaciones web para **enviar y recibir información** entre el cliente (como un navegador) y el servidor (como una API).

📦 ¿Qué significa "ligero"?

Que es fácil de leer por humanos y fácil de procesar por máquinas.

Arreglos Json

Ejemplo de arreglo con objetos:

```json
[
  {
    "id": 1,
    "nombre": "David",
    "edad": 41
  },
  {
    "id": 2,
    "nombre": "Ana",
    "edad": 35
  }
]
```

2. Estructura JSON

La estructura de JSON se basa en **pares clave-valor** dentro de objetos y/o listas de elementos. Las reglas principales son:

- Los objetos se encierran en **llaves `{}`**.
- Los arreglos se encierran en **corchetes `[]`**.
- Las claves siempre son **cadenas entre comillas dobles**.
- Los valores pueden ser: `string`, `number`, `boolean`, `null`, `object`, `array`.

Ejemplo de estructura JSON:

```json
{
  "usuario": {
    "id": 1,
    "nombre": "David",
    "activo": true,
    "roles": ["admin", "editor"],
    "perfil": {
      "ciudad": "Popayán",
      "pais": "Colombia"
    }
  }
}
```

Arreglos Json

3. Validador JSON

Un **validador JSON** es una herramienta que permite comprobar si un JSON está **bien formado** (sintaxis correcta) y opcionalmente si cumple con un **esquema de validación** (estructura esperada).

👉 Ejemplos de validadores en línea:

- https://jsonlint.com/
- https://www.jsonschemavalidator.net/

👉 También existen librerías para validación en distintos lenguajes:

- **JavaScript:** `ajv`, `joi`
- **Python:** `jsonschema`
- **PHP:** `justinrainbow/json-schema`

Ejemplo de esquema de validación (JSON Schema):

```
{
  "type": "object",
  "properties": {
    "id": { "type": "integer" },
    "nombre": { "type": "string" },
    "edad": { "type": "integer", "minimum": 0 }
  },
  "required": ["id", "nombre"]
}
```

Arreglos Json

Este esquema valida que el JSON sea un **objeto** con:

- `id` obligatorio de tipo entero
- `nombre` obligatorio de tipo string
- `edad` opcional, pero si existe debe ser número entero mayor o igual a 0.

En resumen:

- Un **arreglo JSON** es una lista de valores dentro de `[]`.
- La **estructura JSON** se basa en objetos `{}` y arreglos `[]` con pares clave-valor.
- Un **validador JSON** revisa que el JSON sea válido y que cumpla una estructura definida.

Ejemplo Basico

Recibir datos (POST)

```
app.use(express.json()); // middleware para leer JSON

app.post('/usuarios', (req, res) => {
  const usuario = req.body;
  res.send(`Usuario recibido: ${usuario.nombre}`);
});
```

Usa Postman o Insomnia para hacer peticiones POST.

El método **POST** es uno de los principales **métodos HTTP** usados para **enviar datos al servidor**, especialmente cuando estás creando o registrando nueva información.

📫 ¿Qué significa POST?

`POST` = **enviar datos** al servidor para que los **procese o almacene**.

🔧 ¿Dónde se usa?

En formularios, APIs, sistemas de login, registro, creación de productos, etc.

Ejemplos:

- Enviar un formulario de contacto.
- Crear un nuevo usuario.
- Subir un archivo.

POSTMAN

Postman es una herramienta gratuita (con opciones premium) que te permite **probar y desarrollar APIs fácilmente**. Es muy usada por desarrolladores backend y frontend para asegurarse de que los servicios funcionen correctamente.

¿Qué es Postman?

Postman es una aplicación que simula peticiones HTTP (GET, POST, PUT, DELETE, etc.) hacia un servidor o API, sin necesidad de programar una interfaz o frontend.

POSTMAN

¿Para qué sirve Postman?

Función	¿Para qué se usa?
📬 Enviar peticiones HTTP	Probar endpoints de APIs (`GET`, `POST`, `PUT`, `DELETE`, etc.)
📄 Enviar datos	Enviar **JSON, formularios o archivos** al backend fácilmente
📊 Ver respuestas del servidor	Ver lo que responde tu servidor: datos, errores, códigos de estado -200404etc.)
📦 Probar APIs sin frontend	No necesitas crear botones o formularios solo para hacer pruebas
🧪 Automatizar pruebas	Puedes crear tests automáticos para verificar el comportamiento de una API
🧾 Documentar APIs	Crear colecciones y compartirlas como documentación técnica

POSTMAN

¿Cómo se usa?

1. **Instala Postman** desde: https://www.postman.com/downloads/
2. Abre Postman y crea una **nueva petición**.
3. Elige el método HTTP: `GET`, `POST`, etc.
4. Escribe la **URL del endpoint**, por ejemplo:
   `http://localhost:3000/usuario`
5. Si es una petición `POST` o `PUT`, ve a la pestaña "**Body**" → "**raw**" y selecciona `JSON`.
6. Escribe tu JSON de prueba:

```
{
  "nombre": "David",
  "edad": 41
}
```

POSTMAN

Ejemplo visual

- Método: `POST`
- URL: `http://localhost:3000/usuario`
- Body (JSON):

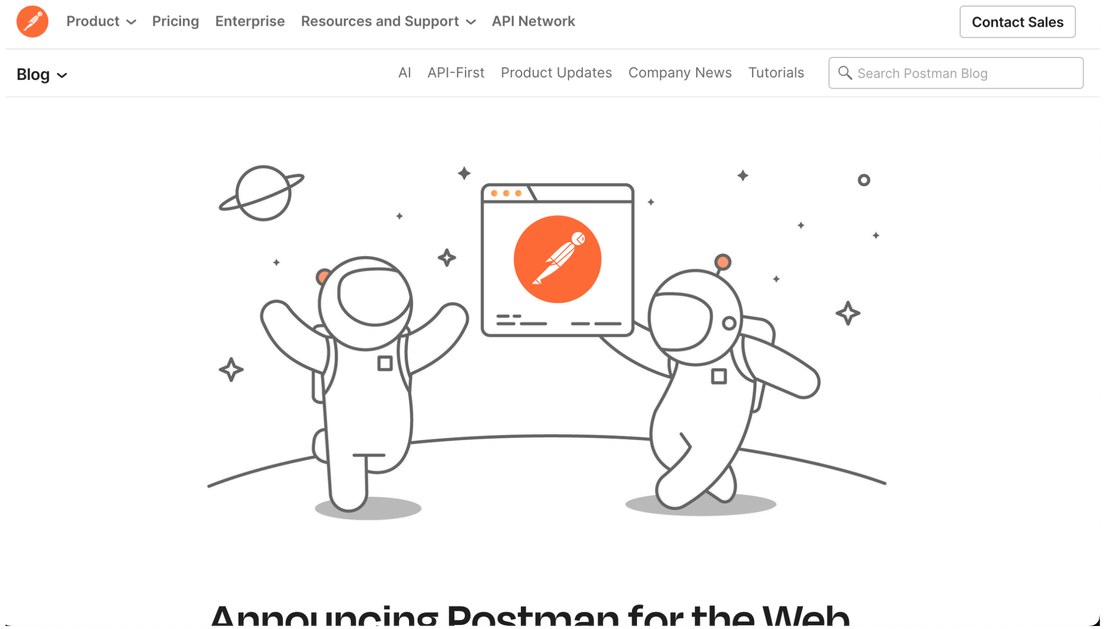
```
{
  "nombre": "Ana",
  "correo": "ana@correo.com"
}
```

En resumen

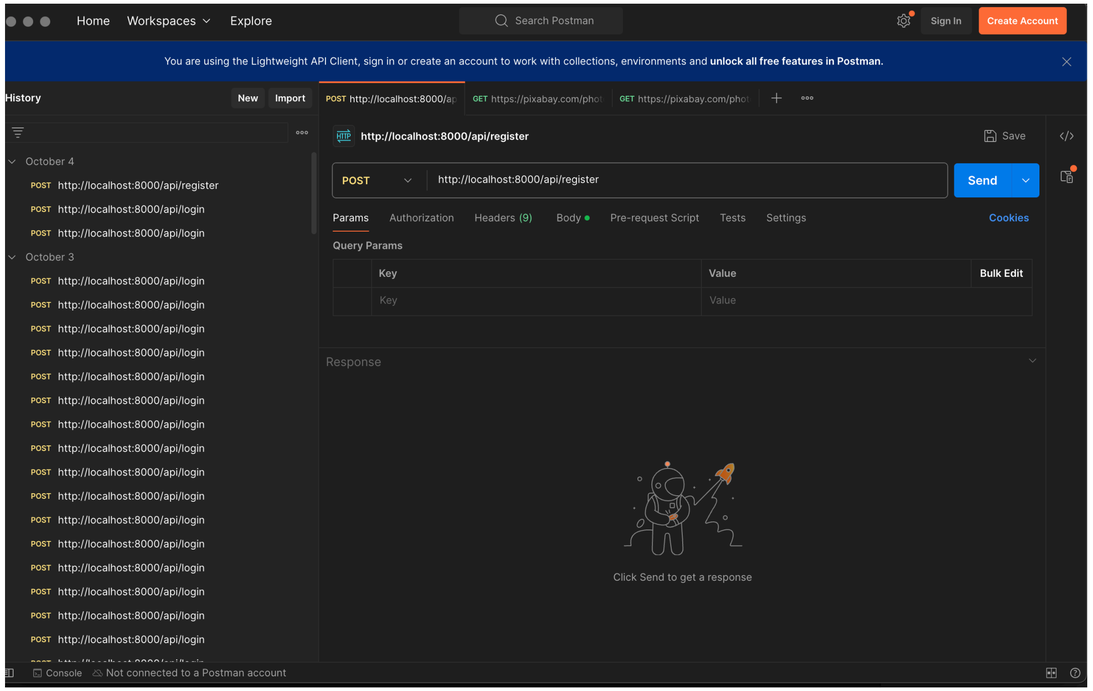
Postman es una herramienta de escritorio para enviar peticiones HTTP a APIs, ideal para probar y depurar servidores, sin necesidad de crear interfaces web.

POSTMAN

Web



Escritorio



MIGRACIONES

Las **migraciones** son **archivos que describen los cambios que deben hacerse en la base de datos**.
Funcionan como un **control de versiones**, pero para tablas, columnas, relaciones y estructuras internas de la base de datos.

Son como un “historial de cambios” que:

- Registra cuándo se creó una tabla
- Cuándo se agregó, cambió o eliminó una columna
- Cuándo se definieron llaves foráneas
- Cuándo se modificaron índices
- Etc.

Cada migración representa un cambio pequeño y controlado.

MIGRACIONES

¿Para qué sirven las migraciones?

✔ 1. Para crear y modificar tablas de forma controlada

En lugar de escribir SQL a mano, las migraciones permiten hacerlo desde código estructurado.

Sirven para:

- Crear tablas
- Agregar o quitar columnas
- Modificar tipos de datos
- Crear claves foráneas
- Crear índices

Todo queda registrado.

MIGRACIONES

¿Para qué sirven las migraciones?

✔ 1. Para crear y modificar tablas de forma controlada

En lugar de escribir SQL a mano, las migraciones permiten hacerlo desde código estructurado.

Sirven para:

- Crear tablas
- Agregar o quitar columnas
- Modificar tipos de datos
- Crear claves foráneas
- Crear índices

Todo queda registrado.

✔ 2. Para mantener sincronizadas las bases de datos del equipo

Cuando varias personas trabajan en un proyecto, cada una puede ejecutar:

```
prisma/schema.prisma
```

Y todos tendrán **exactamente la misma estructura** de base de datos.

Sin migraciones sería un caos.

MIGRACIONES

✔ 3. Para llevar un historial de cambios (versionado)

Cada migración queda guardada en un archivo con su fecha y nombre.

Eso permite:

- Ver quién cambió qué
- Cuándo se creó una tabla
- Qué campos se agregaron
- Qué modificaciones hubo

Igual que Git controla tu código, las migraciones controlan tu base de datos.

✔ 4. Para revertir cambios fácilmente

Si algo sale mal, puedes **revertir**:

```
npx prisma migrate reset
```

Este comando:

- Borra toda la base de datos
- Recrea la BD desde cero
- Aplica todas las migraciones en orden
- Vuelve el proyecto al estado inicial

Ideal cuando cometiste errores serios en las migraciones.

⚠ **BORRA LOS DATOS**, pero NO las migraciones.

BASES DE DATOS

1. Sequelize (SQL)

- Soporta **MySQL, PostgreSQL, MariaDB, SQLite y MSSQL**.
- Permite definir modelos en JavaScript/TypeScript.
- Tiene migraciones, validaciones y asociaciones entre tablas.

2. TypeORM (SQL, más usado con TypeScript)

- Similar a Sequelize, pero con más enfoque en **decoradores** y clases.
- Compatible con MySQL, PostgreSQL, MariaDB, SQLite, Oracle, etc.
- Ideal si usas **TypeScript**.

3. Prisma (moderno y popular 🚀)

- Compatible con **PostgreSQL, MySQL, SQLite, SQL Server, MongoDB**.
- Tiene un **generador de clientes** que facilita las consultas.
- Muy usado en proyectos modernos con Node.js + Express.

4. Mongoose (MongoDB)

- Si tu base de datos es **NoSQL (MongoDB)**, el más usado es **Mongoose**.
- Permite definir esquemas y modelos con facilidad.

PRISMA

PASO 1: Crear proyecto e instalar Prisma

mkdir prisma-mysql

cd prisma-mysql

npm init -y

npm install prisma --save-dev

npm install @prisma/client

PASO 2: Inicializar Prisma

npx prisma init

Esto crea:

- carpeta `prisma/`
- archivo `schema.prisma`
- archivo `.env`

PASO 3: Configurar MySQL en el archivo `.env`

Reemplaza la variable por TU conexión MySQL.

Ejemplo si usas **MySQL local**:

```
DATABASE_URL="mysql://root:password@localhost:3306/prisma_actividad"
```

⚠️ Cambia:

- **root** → tu usuario
- **password** → tu contraseña
- **prisma_actividad** → nombre de tu base de datos

Debes crear la base de datos en MySQL ANTES de migrar:

PRISMA

PASO 4: Configurar MySQL en `schema.prisma`

Abre `prisma/schema.prisma` y ajusta el datasource:

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}
```

PASO 5: Crear dos tablas (models)

En el mismo archivo `schema.prisma`, debajo de lo anterior, define dos modelos y su relación.

Ejemplo profesional:

```
model Usuario {
  id           Int           @id @default(autoincrement())
  nombre       String
  email        String        @unique
  publicaciones Publicacion[]
}

model Publicacion {
  id        Int      @id @default(autoincrement())
  titulo    String
  contenido String
  usuario   Usuario  @relation(fields: [usuarioId], references: [id])
  usuarioId Int
}
```

PRISMA

PASO 6: Ejecutar la migración hacia MySQL

Ahora sí ejecutamos migración:

```
npx prisma migrate dev --name crear_usuarios_y_publicaciones
```

Relaciones entre Modelos

En **Prisma**, una relación **uno a muchos (1:N)** se representa con:

1. En el lado **1** → una lista `[]`.
2. En el lado **N** → una clave foránea (`foreign key`) con `@relation`.

```
model Torneo {
  id      Int     @id @default(autoincrement())
  nombre  String
  grupos  Grupo[]   // 👈 0..N grupos
}

model Grupo {
  id        Int     @id @default(autoincrement())
  nombre    String
  torneoId  Int?        // 👈 Clave foránea opcional
  torneo    Torneo?     @relation(fields: [torneoId], references: [id])
}
```

```
model Disciplina {
  id       Int      @id @default(autoincrement())
  nombre   String
  torneos  Torneo[]   // 👈 Lado "1", tiene muchos torneos
}

model Torneo {
  id           Int        @id @default(autoincrement())
  nombre       String
  disciplinaId Int        // 👈 Foreign key
  disciplina   Disciplina @relation(fields: [disciplinaId], references: [id])
}
```

En Prisma, cuando hablamos de "**de 0 a muchos**" realmente estamos hablando del mismo concepto que **1:N**, pero con la diferencia de que puede ser **opcional** (es decir, una entidad puede tener **ninguno o varios registros relacionados**).

Diferencias:

- `Grupo.torneoId Int?` → el `?` lo hace **opcional**, así que puede existir un `Grupo` sin torneo asociado.
- `Torneo.grupos Grupo[]` → una lista que puede estar vacía (`[]`) o tener varios registros.

BASES DE DATOS

En **Prisma**, una relación **de muchos a muchos (N:M)** se representa con un arreglo (`[]`) en ambos modelos.
Prisma automáticamente crea la **tabla intermedia** (join table) por ti.

```
model Equipo {
  id       Int       @id @default(autoincrement())
  nombre   String
  torneos  Torneo[]  // 👈 muchos torneos
}

model Torneo {
  id      Int      @id @default(autoincrement())
  nombre  String
  equipos Equipo[] // 👈 muchos equipos
}
```

Variación con tabla intermedia explícita

Si la relación N:M tiene atributos extra (por ejemplo, **fecha de inscripción** de un equipo en un torneo), ya no puedes usar la relación implícita.
Debes crear un **modelo intermedio** manual: