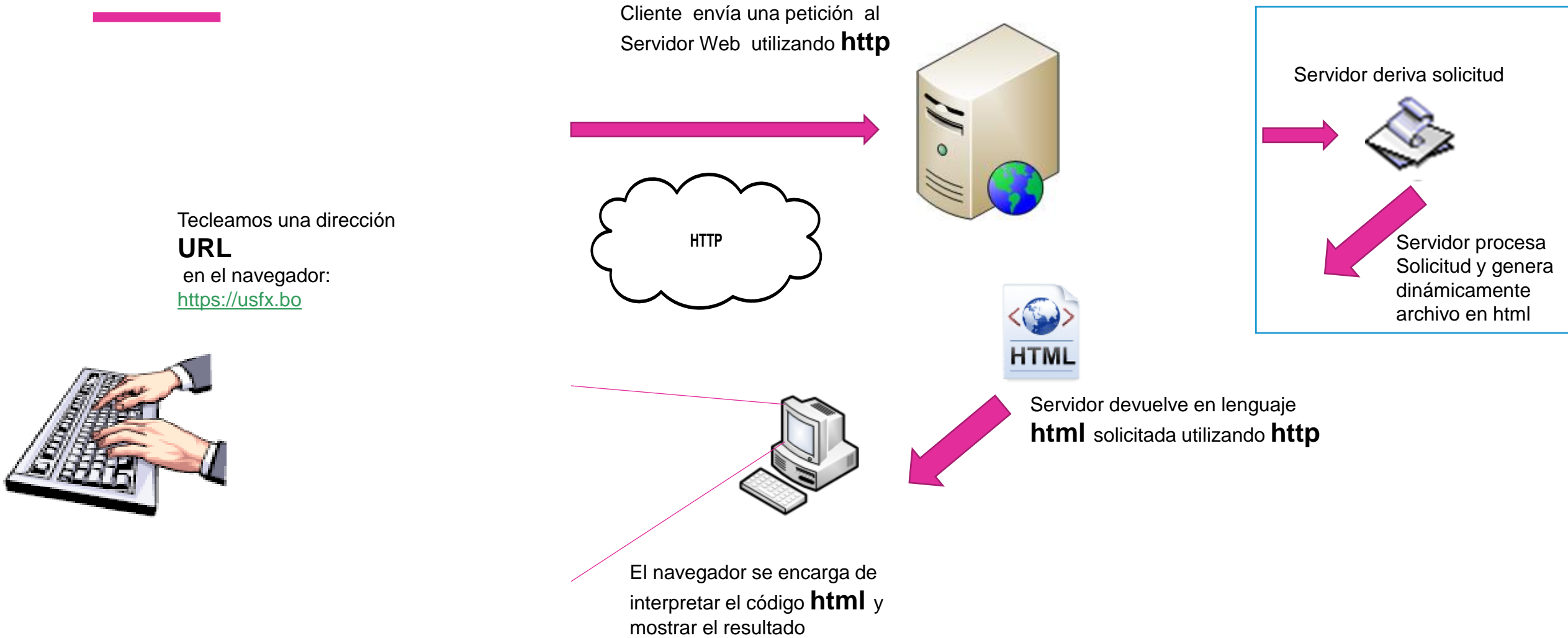




Contenido

- ✓ Introducción a Node.js
- ✓ Creación de aplicaciones web con Express.
- ✓ Acceso a Base de Datos
- ✓ Mongo DB
- ✓ WebSockets

PROCESAMIENTO DEL LADO DEL SERVIDOR





¿Qué es Node.js?

- Node.js es un entorno de ejecución de JavaScript que permite ejecutar código JavaScript fuera del navegador, del lado del servidor
- Creado en 2009 por Ryan Dahl, su objetivo principal era **ofrecer una manera eficiente y escalable de manejar aplicaciones web y servidores** que requieren interacciones rápidas y manejo intensivo de datos en tiempo real.
- Node.js utiliza el motor V8 de Google Chrome, que es el responsable de compilar JavaScript directamente a código máquina, lo que lo hace extremadamente rápido.



Características clave

- **Basado en eventos y no bloqueante:**
 - Node.js utiliza un modelo de entrada/salida no bloqueante, lo que significa que no espera a que una tarea termine para empezar otra. Esto lo hace ideal para aplicaciones en tiempo real como chats y juegos multijugador.
- **Escalabilidad:**
 - Su arquitectura basada en eventos lo hace muy eficiente para manejar grandes cantidades de solicitudes simultáneamente.
- **Uso de JavaScript en el servidor:**
 - Los desarrolladores pueden usar el mismo lenguaje tanto en el cliente como en el servidor, lo que facilita el desarrollo full-stack.
- **Gran ecosistema (NPM):**
 - Node.js cuenta con el Node Package Manager (NPM), que tiene una biblioteca masiva de módulos y paquetes reutilizables para prácticamente cualquier funcionalidad que se necesite.



Historia de Node.js

- 2009: Creado por Ryan Dahl, quien quiso mejorar los servidores web tradicionales que eran bloqueantes y poco eficientes.
- 2010: Lanzamiento de NPM, que se convirtió en un punto clave para la adopción masiva de Node.js.
- 2015: Node.js y io.js (un fork) se fusionaron bajo la Fundación Node.js, mejorando la estabilidad y el soporte.
- Hoy en día: Node.js es usado por gigantes tecnológicos como Netflix, PayPal, LinkedIn, y Walmart para manejar millones de usuarios al mismo tiempo.

Ventajas de Node.js

1

Un solo lenguaje:

- **JavaScript en el frontend y backend:** **Node.js** permite utilizar JavaScript tanto en el lado del cliente como en el servidor. Esto simplifica el desarrollo, ya que los desarrolladores pueden utilizar un solo lenguaje para toda la aplicación, lo que agiliza el proceso y reduce la curva de aprendizaje.
- **Mayor coherencia:** Al utilizar el mismo lenguaje en ambos extremos, se facilita la comunicación entre el cliente y el servidor y se reduce la posibilidad de errores.

2

Alto rendimiento y escalabilidad:

- **Modelo de E/S no bloqueante:** Node.js utiliza un modelo de E/S no bloqueante y basado en eventos, lo que significa que puede manejar muchas conexiones concurrentes sin ralentizar el servidor. Esto lo hace ideal para aplicaciones en tiempo real, como chats o juegos en línea.
- **Microservicios:** Node.js se adapta muy bien a la arquitectura de microservicios, lo que permite escalar las diferentes partes de una aplicación de forma independiente.

3

Gran ecosistema y comunidad:

- **npm (Node Package Manager):** Node.js cuenta con un vasto ecosistema de paquetes y módulos disponibles a través de npm, lo que permite a los desarrolladores reutilizar código y acelerar el desarrollo.
- **Comunidad activa:** La comunidad de Node.js es muy grande y activa, lo que significa que hay una gran cantidad de recursos, tutoriales y soporte disponible.

4

Desarrollo rápido y ágil:

- **Ciclo de desarrollo rápido:** Node.js facilita un ciclo de desarrollo rápido y ágil, gracias a su naturaleza ligera y a la facilidad de crear prototipos.
- **Hot reloading:** Muchas herramientas de desarrollo permiten recargar automáticamente el servidor cuando se realizan cambios en el código, lo que agiliza el proceso de desarrollo.

Ventajas Node.js


- **Ideal para aplicaciones en tiempo real:**
 - **WebSockets:** Node.js es excelente para aplicaciones que requieren comunicación bidireccional en tiempo real, como chats, aplicaciones de colaboración y juegos en línea.
 - **Push notifications:** Node.js facilita la implementación de notificaciones push, lo que permite a las aplicaciones enviar actualizaciones a los usuarios en tiempo real.
- **Fácil de aprender:**
- **Curva de aprendizaje suave:** Si ya conoces JavaScript, aprender Node.js será relativamente sencillo, ya que se basa en los mismos conceptos fundamentales.



Instalación

- Descargar el instalador:
- Visitar la página oficial de Node.js (<https://nodejs.org/>) y descarga el instalador correspondiente a tu sistema operativo (Windows, macOS o Linux). Es mejor trabajar con las versiones LTE
- Ejecutar el instalador:
- Seguir las instrucciones del instalador. Por lo general, solo tienes que hacer clic en "Siguiente" varias veces. Asegúrate de seleccionar las opciones que desees, como agregar Node.js a las variables de entorno.
- Verifica la instalación:
- Abre una terminal o línea de comandos y ejecuta el siguiente comando:

```
node -v  
npm -v
```

Hola Mundo en Node.js

- Crea un archivo JavaScript:
 - Utiliza un editor de texto como Visual Studio Code, Sublime Text o cualquier otro para crear un nuevo archivo con extensión .js. Por ejemplo, hola_mundo.js.2.
- Escribe el código:
 - Abre el archivo y escribe el siguiente código:

```
console.log('¡Hola, mundo desde Node.js!');
```

- Ejecuta el código:
 - En la terminal, navega hasta la carpeta donde guardaste el archivo y ejecuta el siguiente

```
node hola_mundo.js
```

- Deberías ver el mensaje "¡Hola, mundo desde Node.js!" impreso en la consola

Crear Servidores Web

- ¿Qué es un servidor web? Es un programa que escucha por solicitudes HTTP y responde a ellas.
- Node.js, gracias al módulo http, nos permite crear servidores web de forma sencilla.

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('¡Hola desde mi primer servidor Node.js!');
});

server.listen(port, hostname, () => {
  console.log(`Servidor corriendo en http://${hostname}:${port}/`);
});
```



Manejo de Archivos

Crear un archivo

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

const data = 'Esto se escribirá en el archivo';

fs.writeFile('miArchivo.txt', data, (err) => {
  if (err) throw err;
  console.log('El archivo fue creado con éxito!');
});
```

Leer un archivo

```
fs.readFile('miArchivo.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Manejo de la Asincronia

- Es una de sus características más poderosas. Esto significa que Node.js puede manejar múltiples operaciones al mismo tiempo sin bloquear la ejecución del programa.

puede "atender" a múltiples solicitudes al mismo tiempo, sin esperar a que una termine para comenzar la siguiente

Código

```
console.log('Inicio del programa');

// Simulando una tarea que tarda 2 segundos
setTimeout(() => {
  console.log('Tarea 1 completada');
}, 2000);

// Simulando una tarea que tarda 1 segundo
setTimeout(() => {
  console.log('Tarea 2 completada');
}, 1000);

console.log('Fin del programa');
```

Resultado

```
Inicio del programa
Fin del programa
Tarea 2 completada
Tarea 1 completada
```



¿Por qué el orden es diferente al que esperábamos?

- **Ejecución inmediata:** La línea `console.log('Fin del programa');` se ejecuta inmediatamente después de programar las tareas asíncronas, ya que no depende de su resultado.
- **Callbacks:** Las funciones dentro de `setTimeout` son callbacks. Se ejecutan después de que el tiempo especificado haya transcurrido.



¿Como funciona esto en Node.js?

- Node.js utiliza un bucle de eventos. Cuando se recibe una solicitud, se agrega un evento al bucle. El bucle procesa estos eventos de manera no bloqueante, lo que permite que Node.js maneje múltiples solicitudes al mismo tiempo.

Ventajas de la asincronía en Node.js

Alto rendimiento: Node.js puede manejar un gran número de conexiones concurrentes.

Escalabilidad: Las aplicaciones Node.js pueden escalar fácilmente para manejar cargas de trabajo más grandes.

Ideal para aplicaciones en tiempo real: Chats, juegos en línea, etc.

Callbacks

- **Concepto:** Son funciones que se pasan como argumentos a otras funciones y se ejecutan cuando se completa una operación asíncrona.

```
fs.readFile('miArchivo.txt', 'utf8', (err, data) => {  
  if (err) {  
    console.error(err);  
  } else {  
    console.log(data);  
  }  
});
```

Ventajas

Simple de entender y utilizar.

Desventajas

Puede llevar a un "callback hell" cuando se anidan muchos callbacks, dificultando la lectura y el mantenimiento del código.

Promesas

- **Concepto:** Representan el eventual resultado de una operación asíncrona, ya sea un valor exitoso o un error.

```
fs.promises.readFile('miArchivo.txt', 'utf8')
  .then(data => {
    console.log(data);
  })
  .catch(err => {
    console.error(err);
  });
```

Ventajas

Más legible y manejable que los callbacks, permite encadenar operaciones de forma más clara.

Desventajas

Puede requerir un poco más de código inicial para configurar las promesas.

Async/Await

- **Concepto:** Sintaxis de alto nivel que hace que el código asíncrono se parezca más al código síncrono.

```
async function leerArchivo() {  
  try {  
    const data = await  
    fs.promises.readFile('miArchivo.txt', 'utf8');  
    console.log(data);  
  } catch (err) {  
    console.error(err);  
  }  
}
```

Ventajas

Código más limpio y fácil de entender, se parece mucho al código síncrono.

Desventajas

Requiere que la función sea declarada como async.



¿Cuándo usar cada uno?

- **Callbacks:** Ideal para operaciones simples o cuando se trabaja con código legado.
- **Promesas:** Perfecto para operaciones más complejas y encadenamiento de promesas.
- **Async/Await:** Para un código más legible y parecido al síncrono, especialmente en funciones asíncronas más largas.

Express

- **Definición:** Express es un framework minimalista para Node.js que facilita la creación de aplicaciones web y APIs.
- Se construye sobre las capacidades de Node.js, proporcionando una capa adicional de abstracción que simplifica el desarrollo de servidores web.

¿Qué permite hacer Express.js?

- **Crear rutas:** Definir las diferentes URLs que tu aplicación puede manejar y las acciones a realizar cuando se acceden a ellas (por ejemplo, mostrar una página, procesar un formulario).
- **Manejar solicitudes HTTP:** Procesar las solicitudes HTTP entrantes (GET, POST, PUT, DELETE, etc.) y generar las respuestas correspondientes.
- **Utilizar middleware:** Agregar funcionalidades adicionales a las solicitudes, como autenticación, autorización, registro de logs, etc.
- **Renderizar vistas:** Generar páginas HTML dinámicas a partir de plantillas.
- **Gestionar errores:** Capturar y manejar errores de forma centralizada.

¿Cómo se instala Express.js?

- Crea un nuevo directorio para tu proyecto e ingresa a el

```
mkdir mi-proyecto-express  
cd mi-proyecto-express
```

- Inicializa un proyecto npm

```
npm init -y
```

- Instala Express

```
npm install express
```



Ejemplo básico

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

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

app.listen(port, () => {
  console.log(`Servidor escuchando en el puerto ${port}`);
});
```

Manejo diferentes rutas

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

// Middleware para manejar datos en formato JSON
app.use(express.json());


// Rutas básicas
app.get('/', (req, res) => {
  res.send('Bienvenido a mi aplicación con Express');
});

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



Conceptos Clave y Consideraciones

- **Rutas:** Definen las diferentes direcciones URL que tu aplicación puede manejar y las acciones asociadas a cada una.
- **Middleware:** Funciones que se ejecutan antes de llegar a la ruta de destino. Se utilizan para tareas como autenticación, autorización, registro de logs, etc.
- **Vistas:** Plantillas que se utilizan para generar HTML dinámico.
- **Modelos:** Representan los datos de tu aplicación y se utilizan para interactuar con bases de datos.
- **Controladores:** Manejan las solicitudes HTTP y coordinan la lógica de la aplicación.



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

// Middleware
app.use(express.json()); // Para parsear JSON
app.use(express.urlencoded({ extended: false })); // Para
parsear formularios

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

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

Creación de rutas:

- GET: Para obtener datos.
- POST: Para enviar datos al servidor.
- PUT: Para actualizar datos.
- DELETE: Para eliminar datos.
- Otros: OPTIONS, HEAD, PATCH, etc

Middleware:

- Built-in: Express proporciona middleware para parsear JSON, formularios, etc.
- Custom: Puedes crear tu propio middleware para tareas personalizadas.

Ejemplo básico de manejo de formulario

```
const express = require('express');
const app = express();
const port = 3000;
// Middleware para parsear los datos del formulario
app.use(express.urlencoded({ extended: false }));
// Ruta para mostrar el formulario
app.get('/', (req, res) => {
  res.send(`
    <form method="POST" action="/calcular">
      <label for="num1">Número 1:</label>
      <input type="number" id="num1" name="num1">
      <br>
      <label for="num2">Número 2:</label>
      <input type="number" id="num2" name="num2">
      <br>
      <button type="submit">Calcular suma</button>
    </form>
  `);
});
// Ruta para calcular la suma
app.post('/calcular', (req, res) => {
  const num1 = parseInt(req.body.num1);
  const num2 = parseInt(req.body.num2);
  const suma = num1 + num2;
  res.send(`La suma de ${num1} y ${num2} es: ${suma}`);
});
app.listen(port, () => {
  console.log(`Servidor escuchando en el puerto ${port}`);
});
```

Devolver pagina estática

server.js

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

const app = express();

// Configurar la carpeta 'public' como el directorio de
archivos estáticos
app.use(express.static(path.join(__dirname, 'public')));

// Ruta para el home
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public',
'bienvenidos.html'));
});

// Iniciar el servidor
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Servidor corriendo en
http://localhost:${PORT}`);
});
```

Bienvenido.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Bienvenidos</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>¡Bienvenido a mi página!</h1>
  <p>Esta es una página HTML servida por Node.js.</p>
</body>
</html>

body {
  font-family: Arial, sans-serif;
  text-align: center;
  background-color: #f3f3f3;
  margin: 0;
  padding: 20px;
}

h1 {
```

- Utiliza motores de plantillas como Pug, EJS o Handlebars para generar HTML dinámico.

Vistas

```
my-app/  
  app.js  
  views/  
    index.ejs  
    resultado.ejs
```

```
const express = require('express');  
const app = express();  
const port = 3000;  
// Configuramos EJS como motor de plantillas  
app.set('views', './views');  
app.set('view engine', 'ejs');  
// Middleware para parsear datos de formularios  
app.use(express.urlencoded({ extended: false }));  
// Ruta para mostrar el formulario  
app.get('/', (req, res) => {  
  res.render('index');  
});  
  
// Ruta para calcular la suma y mostrar el resultado  
app.post('/calcular', (req, res) => {  
  const num1 = parseInt(req.body.num1);  
  const num2 = parseInt(req.body.num2);  
  const suma = num1 + num2;  
  res.render('resultado', { suma });  
});  
  
app.listen(port, () => {  
  console.log(`Servidor escuchando en el puerto ${port}`);  
});
```

index.ejs

```
<form method="POST" action="/calcular">  
  <label for="num1">Número 1:</label>  
  <input type="number" id="num1"  
  name="num1">  
  <br>  
  <label for="num2">Número 2:</label>  
  <input type="number" id="num2"  
  name="num2">  
  <br>  
  <button type="submit">Calcular</button>  
</form>
```

resultado.ejs

```
<h1>El resultado de la suma es: <%= suma
```

Renderiaziones mas complejas

```
my-app/  
  app.js  
  views/  
    index.ejs
```

```
app.get('/', (req, res) => {  
  const productos = [  
    { nombre: 'Manzana', precio: 1.5 },  
    { nombre: 'Banana', precio: 0.8 },  
    { nombre: 'Naranja', precio: 1.2 }  
  ];  
  
  res.render('index', { productos });  
});
```

```
<table>  
  <thead>  
    <tr>  
      <th>Nombre</th>  
      <th>Precio</th>  
    </tr>  
  </thead>  
  <tbody>  
    <% productos.forEach(producto => { %>  
      <tr>  
        <td><%= producto.nombre %></td>  
        <td><%= producto.precio %></td>  
      </tr>  
      <% }); %>  
    </tbody>  
</table>
```

```
my-app/  
  app.js  
  views/  
    index.ejs
```

```
const express = require('express');  
const app = express();  
const port = 3000;  
  
// Configuramos EJS como motor de plantillas  
app.set('views', './views');  
app.set('view engine', 'ejs');  
  
// Middleware para parsear datos de formularios  
app.use(express.urlencoded({ extended: false }));  
  
// Ruta para mostrar el formulario  
app.get('/', (req, res) => {  
  res.render('index');  
});  
  
// Ruta para generar la tabla  
app.post('/generarTabla', (req, res) => {  
  const numFilas = parseInt(req.body.filas);  
  const numColumnas = parseInt(req.body.columnas);  
  
  // Generar la tabla en formato HTML  
  let tabla = '<table>';  
  for (let i = 0; i < numFilas; i++) {  
    tabla += '<tr>';  
    for (let j = 0; j < numColumnas; j++) {  
      tabla += '<td>Celda</td>';  
    }  
    tabla += '</tr>';  
  }  
  tabla += '</table>';  
  
  res.render('resultado', { tabla });  
});  
  
app.listen(port, () => {  
  console.log(`Servidor escuchando en el puerto ${port}`);  
});
```

```
<form method="POST" action="/generarTabla">  
  <label for="filas">Número de filas:</label>  
  <input type="number" id="filas" name="filas">  
  <br>  
  <label for="columnas">Número de columnas:</label>  
  <input type="number" id="columnas" name="columnas">  
  <br>  
  <button type="submit">Generar Tabla</button>  
</form>
```

```
<%= tabla %>
```

EJS

- Sintaxis básica

```
<p>Hola, <%= nombre  
%>!</p>
```

- Insertar variables: Para insertar el valor de una variable en el HTML, se utilizan las etiquetas <%%>


Control de flujo

Puedes utilizar estructuras de control como if, else y for para mostrar contenido condicionalmente o iterar sobre arreglos.

Expresiones: Puedes realizar operaciones matemáticas y lógicas dentro de las etiquetas <% %>

```
<p>El resultado de la suma es: <%= numero1 + numero2  
%>.</p>
```

```
<% if (usuarios.length > 0) { %>  
  <ul>  
    <% usuarios.forEach(usuario => { %>  
      <li><%= usuario.nombre %></li>  
    <% }>); %>  
  </ul>  
  <% } else { %>  
    <p>No hay usuarios registrados.</p>  
  <% } %>
```



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

app.set('views', './views');
app.set('view engine', 'ejs');

app.get('/', (req, res) => {
  const productos = [
    { nombre: 'Manzana', precio: 1.5 },
    { banana: 'Banana', precio: 0.8 }
  ];
  res.render('index', { productos });
});

app.listen(port, () => {
  console.log(`Servidor escuchando en el puerto ${port}`);
});
```

```
<h1>Productos</h1>
<ul>
  <% productos.forEach(producto => { %>
    <li><%= producto.nombre %> - $<%=
producto.precio %></li>
  <% }); %>
</ul>
```

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

// Configuración de EJS como motor de plantillas
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// Datos dinámicos
const usuarios = [
  { id: 1, nombre: 'Juan', edad: 30 },
  { id: 2, nombre: 'Ana', edad: 25 },
];

// Ruta dinámica
app.get('/usuarios', (req, res) => {
  res.render('usuarios', { usuarios });
});

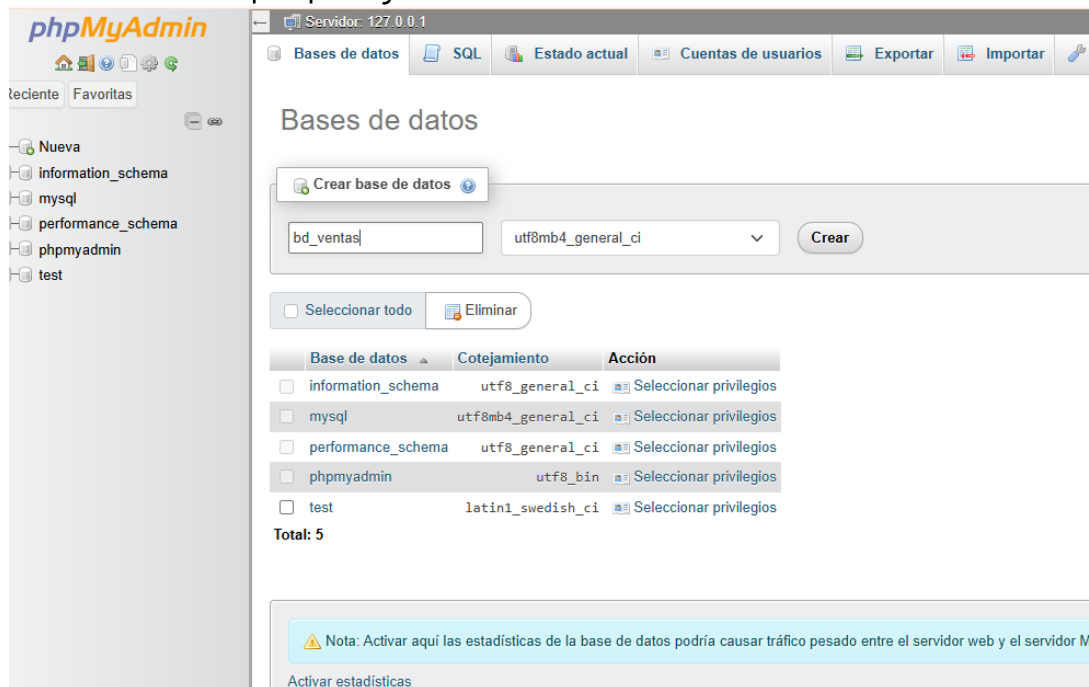
app.listen(3000, () => {
  console.log('Servidor corriendo en http://localhost:3000/');
});
```

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Lista de Usuarios</title>
</head>
<body>
  <h1>Usuarios Registrados</h1>
  <ul>
    <% usuarios.forEach(usuario => { %>
      <li><%= usuario.nombre %> (Edad: <%=
usuario.edad %>)</li>
    <% }); %>
  </ul>
</body>
</html>
```


Uso de Base de Datos Relacionales desde Node

Conexión a la base de datos MySQL

- Desde phpmyadmin crear la base de datos bd_ventas y la tabla productos



- 
- Crea un archivo db.js para manejar la conexión con bd_ventas

Conexión a la base de datos MySQL

```
const mysql = require('mysql2');

// Configuración de la conexión
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',      // Usuario de tu base de datos
  password: '',      // Contraseña de tu base de datos
  database: 'bd_ventas' // Nombre de la base de datos
});

// Conexión a la base de datos
connection.connect((err) => {
  if (err) {
    console.error('Error al conectar a la base de datos:',
err);
  }
  return;
```

```
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./db');

const app = express();
app.use(bodyParser.urlencoded({ extended: true })); // Parsear
datos de formularios
app.use(express.static('public')); // Servir archivos estáticos
app.set('view engine', 'ejs'); // Configurar EJS como motor de
plantillas
```

```
  Página principal: listar productos
app.get('/', (req, res) => {
  const query = 'SELECT * FROM Productos';
  db.query(query, (err, results) => {
    if (err) {
      return res.status(500).send(err);
    }
    res.render('index', { productos: results });
  });
});
```

```
// Mostrar formulario para agregar producto
app.get('/add', (req, res) => {
  res.render('add');
});
```

```
// Procesar formulario para agregar producto
app.post('/add', (req, res) => {
  const { Nombre, Precio, Stock } = req.body;
  const query = 'INSERT INTO Productos (Nombre, Precio,
Stock) VALUES (?, ?, ?)';
  db.query(query, [Nombre, Precio, Stock], (err) => {
    if (err) {
      return res.status(500).send(err);
    }
    res.redirect('/');
  });
});
```

```
// Mostrar formulario para editar producto
app.get('/edit/:id', (req, res) => {
  const { id } = req.params;
  const query = 'SELECT * FROM Productos WHERE Id = ?';
  db.query(query, [id], (err, results) => {
    if (err) {
      return res.status(500).send(err);
    }
    res.render('edit', { producto: results[0] });
  });
});
```

Configuración del Servidor

```
// Procesar formulario para editar producto
app.post('/edit/:id', (req, res) => {
  const { id } = req.params;
  const { Nombre, Precio, Stock } = req.body;
  const query = 'UPDATE Productos SET Nombre = ?, Precio = ?, Stock = ? WHERE
Id = ?';
  db.query(query, [Nombre, Precio, Stock, id], (err) => {
    if (err) {
      return res.status(500).send(err);
    }
    res.redirect('/');
  });
});

// Eliminar producto
app.get('/delete/:id', (req, res) => {
  const { id } = req.params;
  const query = 'DELETE FROM Productos WHERE Id = ?';
  db.query(query, [id], (err) => {
    if (err) {
      return res.status(500).send(err);
    }
    res.redirect('/');
  });
});

// Iniciar servidor
app.listen(3000, () => {
  console.log('Servidor corriendo en http://localhost:3000');
});
```

Crear vistas EJS

views/layout.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>CRUD Ventas</title>
  <link rel="stylesheet" href="/styles.css">
</head>
<body>
  <header>
    <h1>Gestión de Productos</h1>
    <nav>
      <a href="/">Inicio</a>
      <a href="/add">Agregar Producto</a>
    </nav>
  </header>
  <main>
    <%- body %>
  </main>
</body>
</html>
```

views/listar.ejs

```
<% include('layout', { body: null }) %>
<h2>Lista de Productos</h2>
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Precio</th>
      <th>Stock</th>
      <th>Acciones</th>
    </tr>
  </thead>
  <tbody>
    <% productos.forEach(producto => { %>
      <tr>
        <td><%= producto.Nombre %></td>
        <td><%= producto.Precio %></td>
        <td><%= producto.Stock %></td>
        <td>
          <a href="/edit/<%= producto.Id %>">Editar</a>
          <a href="/delete/<%= producto.Id %>">Eliminar</a>
        </td>
      </tr>
    <% }) %>
  </tbody>
</table>
```

Crear vistas EJS

views/index.ejs

```
<% include('layout', { body: null }) %>
<h2>Lista de Productos</h2>
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Precio</th>
      <th>Stock</th>
      <th>Acciones</th>
    </tr>
  </thead>
  <tbody>
    <% productos.forEach(producto => { %>
      <tr>
        <td><%= producto.Nombre %></td>
        <td><%= producto.Precio %></td>
        <td><%= producto.Stock %></td>
        <td>
          <a href="/edit/<%= producto.Id %>">Editar</a>
          <a href="/delete/<%= producto.Id %>">Eliminar</a>
        </td>
      </tr>
    <% }) %>
```

views/add.ejs

```
<% include('layout', { body: null }) %>
<h2>Agregar Producto</h2>
<form action="/add" method="POST">
  <label>Nombre: <input type="text" name="Nombre"
required></label>
  <label>Precio: <input type="number" name="Precio"
step="0.01" required></label>
  <label>Stock: <input type="number" name="Stock"
required></label>
  <button type="submit">Guardar</button>
</form>
```

views/edit.ejs

```
<% include('layout', { body: null }) %>
<h2>Editar Producto</h2>
<form action="/edit/<%= producto.Id %>" method="POST">
  <label>Nombre: <input type="text" name="Nombre"
value="<%= producto.Nombre %>" required></label>
  <label>Precio: <input type="number" name="Precio"
value="<%= producto.Precio %>" step="0.01"
required></label>
  <label>Stock: <input type="number" name="Stock"
value="<%= producto.Stock %>" required></label>
  <button type="submit">Actualizar</button>
```



Integrando TypeORM con Node.js y Express

- TypeORM es un ORM (Object-Relational Mapper) popular para TypeScript y JavaScript que facilita la interacción con bases de datos relacionales.
- Al combinarlo con Node.js y Express, podemos crear aplicaciones web más estructuradas y escalables



¿Por qué usar TypeORM?

- **Tipado:** Ofrece un fuerte tipado, lo que reduce errores y mejora la legibilidad del código.
- **Abstracción:** Oculta la complejidad de las consultas SQL, permitiendo trabajar con objetos JavaScript.
- **Relaciones:** Maneja de forma sencilla las relaciones entre entidades (uno a uno, uno a muchos, muchos a muchos).
- **Migraciones:** Permite realizar cambios en la estructura de la base de datos de forma controlada y segura.

Ejemplo

```
import { createConnection } from 'typeorm';
import { User } from './entity/User';
import * as express from 'express';
```

```
createConnection()
  .then(async connection => {
```

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

```
    app.get('/users', async (req, res) => {
      const users = await
connection.manager.find(User);
      res.send(users);
    });
```

```
    app.post('/users', async (req, res) => {
      const user = new User();
      user.name = req.body.name;
      user.email = req.body.email;
      await connection.manager.save(user);
      res.send('Usuario creado');
    });
```


```
router.put('/:id', async (req, res) => {
  try {
    const userRepository = getRepository(User);
    const user = await userRepository.findOne(req.params.id);

    if (!user) {
      return res.status(404).json({ message: 'Usuario no
encontrado' });
    }

    user.name = req.body.name || user.name;
    user.email = req.body.email || user.email;

    await userRepository.save(user);

    res.json(user);
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: 'Error al actualizar el
usuario' });
  }
});
```



```
// Eliminar un usuario
router.delete('/:id', async (req, res) => {
  try {
    const userRepository = getRepository(User);
    const result = await userRepository.delete(req.params.id);

    if (result.affected === 0) {
      return res.status(404).json({ message: 'Usuario no encontrado' });
    }

    res.json({ message: 'Usuario eliminado correctamente' });
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: 'Error al eliminar el usuario' });
  }
});
```



```
import { Entity, PrimaryGeneratedColumn, Column } from  
'typeorm';
```

```
@Entity()  
export class User {  
  @PrimaryGeneratedColumn()  
  id: number;  
  
  @Column()  
  name: string;  
}
```

```
// userRepository.ts  
import { EntityRepository, Repository } from 'typeorm';  
import { User } from './User';
```

```
@EntityRepository(User)  
export class UserRepository extends Repository<User> {}
```

Base de Datos NoSQL



¿Qué son las bases de datos NoSQL?

- Son sistemas de gestión de bases de datos que no se adhieren al modelo relacional tradicional.
- Ofrecen una mayor flexibilidad en la estructura de los datos y en las operaciones que se pueden realizar sobre ellos.
- Son especialmente útiles para manejar grandes volúmenes de datos no estructurados o semiestructurados.



Características principales

- No SQL: No utilizan el lenguaje SQL para realizar consultas.
- Esquema flexible: No requieren un esquema rígido definido previamente, lo que permite adaptarse a cambios en los datos.
- Escalabilidad horizontal: Pueden escalar fácilmente agregando más servidores para manejar cargas de trabajo más grandes.
- Alto rendimiento: Diseñadas para manejar grandes volúmenes de datos y realizar consultas de alta velocidad.
- Modelo de datos: Pueden utilizar diferentes modelos de datos, como documentos, clave-valor, columnas o grafos.



Tipos de bases de datos NoSQL

- Documentales:
 - Almacenan datos en documentos JSON-like.
 - Ejemplos: MongoDB, Couchbase
- Clave-valor:
 - Almacenan datos como pares clave-valor.
 - Ejemplos: Redis, Amazon DynamoDB
- Columnarias:
 - Optimizadas para consultas analíticas y grandes volúmenes de datos.
 - Ejemplos: Cassandra, Hbase
- Gráficas:
 - Modelan datos como nodos y relaciones.
 - Ejemplos: Neo4j, Amazon Neptune

Cuándo utilizar bases de datos NoSQL:

- Grandes volúmenes de datos: Cuando se tienen grandes cantidades de datos que crecen rápidamente.
- Datos no estructurados o semiestructurados: Cuando los datos no se ajustan fácilmente a un esquema relacional.
- Alta disponibilidad: Cuando se requiere un alto nivel de disponibilidad y tolerancia a fallos.
- Escalabilidad horizontal: Cuando se necesita escalar la base de datos agregando más servidores.
- Rendimiento en consultas específicas: Cuando las consultas son muy específicas y se pueden optimizar mejor en un modelo NoSQL.





Ventajas de las bases de datos NoSQL

- Flexibilidad: Fácil adaptación a cambios en los requisitos de los datos.
- Escalabilidad: Capacidad de manejar grandes volúmenes de datos y altas cargas de trabajo.
- Rendimiento: Optimizadas para ciertas cargas de trabajo, como consultas analíticas o de lectura.



Desventajas de las bases de datos NoSQL

- Complejidad: Pueden ser más difíciles de administrar y configurar que las bases de datos relacionales.
- Madurez: Algunas tecnologías NoSQL pueden ser menos maduras que las bases de datos relacionales.
- Falta de estandarización: No existe un estándar único para las bases de datos NoSQL, lo que puede dificultar la migración entre diferentes sistemas.

Mongo db



Qué es MongoDB?

- MongoDB es una base de datos documental, lo que significa que almacena datos en documentos que tienen una estructura similar a **JSON**.
- A diferencia de las bases de datos relacionales tradicionales, MongoDB no requiere un esquema rígido y predefinido para las tablas, lo que te brinda una gran flexibilidad para modelar tus datos.



Características principales de MongoDB

- **Documentos:** Los datos se almacenan en documentos, que son estructuras de datos similares a JSON. Cada documento puede tener diferentes campos y estructuras, lo que facilita la adaptación a diferentes tipos de datos.
- **Colecciones:** Los documentos se agrupan en colecciones, que son similares a las tablas en una base de datos relacional, pero sin un esquema fijo.
- **Esquema dinámico:** MongoDB no impone un esquema estricto, lo que significa que puedes agregar nuevos campos a los documentos en cualquier momento sin tener que modificar la estructura de la colección.
- **Indexación:** Permite crear índices en los campos de los documentos para mejorar el rendimiento de las consultas.
- **Escalabilidad:** MongoDB es altamente escalable, lo que significa que puede manejar grandes volúmenes de datos y un gran número de usuarios. Alta disponibilidad: Ofrece características de replicación y recuperación ante desastres para garantizar la alta disponibilidad de los datos



¿Cuándo usar MongoDB?

- Datos no estructurados o semiestructurados: MongoDB es ideal para almacenar datos que no encajan fácilmente en un esquema relacional rígido, como datos de aplicaciones móviles, registros de eventos o análisis de datos. Aplicaciones de gran escala: Su escalabilidad y rendimiento lo hacen adecuado para aplicaciones que requieren manejar grandes volúmenes de datos y un alto tráfico. Aplicaciones de desarrollo ágil: La flexibilidad de MongoDB permite adaptarse rápidamente a los cambios en los requisitos del proyecto.

CRUD - Create

Las operaciones CRUD son Create, Read, Update y Delete documentos.

- ▶ La operación de creación o inserción añade nuevos documentos a una colección.
- ▶ Importante: si la colección no existe actualmente, las operaciones de inserción crearán la colección.
- ▶ MongoDB proporciona los siguientes métodos para insertar documentos en una colección:

`db.<collection>.insert()`

`db.<collection>.insertOne()`

`db.<collection>.insertMany()`

```
db.users.insertOne(  ← collection
{
  name: "sue",        ← field: value
  age: 26,            ← field: value
  status: "pending"   ← field: value
}                    } document
)
```

- ▶ En MongoDB, las operaciones de inserción se aplican sobre una única colección.
- ▶ Todas las operaciones de escritura en MongoDB **son atómicas** en el nivel de un solo documento.

CRUD - Create

- ▶ `insert()` inserta uno o varios documentos en una colección. Devuelve la descripción de la operación y el número de registros insertados.

```
> db.inventory.insert( { item: "papas", qty: 120, tags: ["verdu"], size: { h: 30, w: 35.5, uom: "cm" } } )
WriteResult({ "nInserted" : 1 })
```

- ▶ `insertOne()` inserta un único documento en una colección y devuelve un documento que incluye el valor de campo `_id` del documento recién insertado.

```
> db.inventory.insertOne(
...   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
... )
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5ec04285534ad19d22230259")
}
```

CRUD - Create

- ▶ `insertMany()` inserta varios documentos en una colección y devuelve un documento que incluye el valor del campo `_id` de los documentos recién insertados.

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5ec044cd534ad19d2223025b"),
    ObjectId("5ec044cd534ad19d2223025c"),
    ObjectId("5ec044cd534ad19d2223025d")
  ]
}
```

- ▶ Las operaciones de inserción crearán la colección si esta no existe.
Cada documento almacenado en una colección requiere de un campo `_id` único que actúa como clave principal.
- ▶ Si el documento nuevo omite dicho campo, el controlador de MongoDB generará automáticamente un **ObjectId** para el campo `_id`.

CRUD - Read

- ▶ Se acceden a datos de una colección, No hay nada similar al JOIN de SQL
- ▶ El método `find`(buscar): devuelve un cursor para recorrer el resultado.

```
> db.<coleccion>.find( <filtros>, <proyecciones> )
```

```
> db.products.find( )
```



```
> db.products.find( {} )
```



```
> db.products.find
```

```
> db.products.find().pretty()
```



- ▶ El método `findOne` (buscar un único resultado): devuelve el primer resultado de la consulta.

```
> db.<coleccion>.findOne( <filtros>, <proyecciones> )
```

CRUD - Read

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
) .limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

Las consultas en MongoDB devuelven una colección de documentos que cumplan una condición determinada.

- Para consultar o leer los documentos, MongoDB proporciona los siguientes métodos:

```
> db.products.find()
```

```
SELECT * FROM products
```

```
> db.products.find( { status: { $in: [ "A", "D" ] } } )
```

```
SELECT * FROM products WHERE status in ("A", "D")
```

```
> db.products.find( { status: "A", qty: { $lt: 30 } } )
```

```
SELECT * FROM products WHERE status = "A" AND qty < 30
```

CRUD - Read

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

El método `db.collection.findOne()` también realiza una operación de lectura para devolver un solo documento. Internamente, el método `db.collection.findOne()` es el método `db.collection.find()` con un límite de 1:

```
> db.products.find( {  
  status: "A",  
  $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]  
} )
```

```
SELECT * FROM products WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

CRUD - Read

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
)
```

← collection
← query criteria
← projection
← cursor modifier

```
> db.<coleccion>.find( { "field1": "value", "field2": "value" } )
```

```
> use fwd
```

```
> db.product.find( { "is_active": true, "weight": 1 } )
```

```
> db.product.find(...) .limit(n) .pretty()      //n = número registros
```

```
> db.product.findOne( { "is_active": true, "weight": 1 } )
```

```
> db.product.findOne(...) .pretty()
```

CRUD - Read

```
...  
"price": 332.20,  
"variants" : {  
  "1" : { "name" : "Blue, Small", "color": "blue" },  
  "2" : { "name" : "Black, Small", "color": "orange" },  
  "3" : { "name" : "Pink, Small", "color": "yellow" }  
},  
"is_active" : true,  
...
```

Para especificar una condición de igualdad en un campo que es un documento incrustado o anidado, use el documento de filtro de consulta {<campo>: <valor>} y asigne a <valor> el documento que quiere buscar o consultar.

```
> db.products.find( { "variants.1": { "name": "Blue, Small", "color": "red" } } )
```

Las coincidencias de igualdad en todo el documento incrustado requieren una coincidencia exacta del documento <valor> especificado, incluido el orden de los campos. Por ejemplo, la siguiente consulta no coincide con ningún documento en la colección de inventario:

```
> db.products.find( { "variants.1": { "color": "red", "name": "Blue, Small" } } )
```

- ▶ Cuando utilice la notación de punto, el campo y el campo anidado deben estar entre comillas:

CRUD - Read

Aportan una nueva dimensión los datos

- ❑ Permiten almacenar "varios registros" dentro de uno.
 - ❑ En bases de datos relacionales necesitaríamos utilizar varias tablas la mayoría de las veces.
- ❑ Esta disposición, complican las operaciones.
 - ❑ Siempre se debe recordar que la unidad básica de trabajo son los documentos.

Notación de puntos (dot notation)

- ❑ Permite acceder a valores que estén dentro de listas o subdocumentos:
 - ❑ Listas: Índice del elemento
- ❑ Subdocumentos:
 - ❑ Clave del elemento

```
"category_ids.0"
```

```
"variants.1.name"
```


CRUD - Read

"variants.1.name"

"category_ids.0"

```
{
  "_id": ObjectId("53f8659a957cfd6a0a8b4595"),
  "name": "The North Face McMurdo II Boot ...",
  "primary_category_id": 8,
  "price": 135.95,
  "variants": {
    "1": {
      "name": "Brown, 7"
    },
    "2": {
      "name": "Black, 7"
    }
  },
  "category_ids": [ 8, 15 ],
  "date_created": ISODate("2014-08-23T09:57:46Z")
}
```

CRUD - Read

Búsquedas en listas

- ❑ Es posible realizar búsquedas exactas o de elementos puntuales
 - ❑ Coincidencia exacta:

```
{ "category_ids": [8, 15] }
```

- ❑ Coincidencia exacta:

```
{ "category_ids": 8 }
```

```
{
  "_id": ObjectId("53f8659a957cfd6a0a8b4595"),
  "name": "The North Face McMurdo II Boot ...",
  "primary_category_id": 8,
  "price": 135.95,
  "variants": {
    "1": {
      "name": "Brown, 7"
    },
    "2": {
      "name": "Black, 7"
    }
  },
  "category_ids": [ 8, 15 ],
  "date_created": ISODate("2014-08-23T09:57:46Z")
}
```

CRUD - Read

Búsquedas en subdocumentos

- ❑ Es posible realizar búsquedas exactas o de elementos puntuales
 - ❑ Coincidencia exacta:

```
{ "variants": {  
  "1" : { "name": "Brown, 7" },  
  "2" : { "name": "Black, 7" }  
}
```

- ❑ Coincidencia exacta:

```
{ "variants.2.name": "Black, 7" }
```

```
{  
  "_id": ObjectId("53f8659a957cfd6a0a8b4595"),  
  "name": "The North Face McMurdo II Boot ...",  
  "primary_category_id": 8,  
  "price": 135.95,  
  "variants": {  
    "1": {  
      "name": "Brown, 7"  
    },  
    "2": {  
      "name": "Black, 7"  
    }  
  },  
  "category_ids": [ 8, 15 ],  
  "date_created": ISODate("2014-08-23T09:57:46Z")  
}
```

CRUD - Read


Obtener subconjunto de resultados

Métodos asociados al cursor

limit (limitar)

- ❑ Obtener como máximo X resultados de una consulta
- ❑ Mejora en rendimiento, evita recorrer resultados extra

skip (saltar)

- ❑ Ignorar los primeros X resultados de una consulta
- ❑  No envía o muestra los resultados ignorados, pero sí los recorre.

```
> db.products.find().skip( 5 ).limit( 5 )
```

CRUD - Read

```
db.library.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

Para especificar la condición de igualdad en un array, se debe usar como <valor> un array exacto al que se quiere buscar, teniendo en cuenta el orden de los elementos.

```
> db.library.find( { tags: ["red", "blank"] } )
```

Si lo que desea es consultar un array que contenga los elementos "rojo" y "en blanco", sin tener en cuenta el orden u otros elementos del array, use el operador [\\$all](#):

```
> db.library.find( { tags: { $all: ["red", "blank"] } } )
```

Para consultar si el campo de array contiene al menos un elemento con el valor especificado, use el filtro {<campo>: <valor>} donde <valor> es el valor del elemento:

```
> db.library.find( { tags: "red" } );
```

CRUD - Read

```
db.library.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

- Para especificar condiciones en los elementos en el campo del array, puede usar operadores de consulta en el documento de filtro de consulta.

El ejemplo anterior, permite consultar todos los documentos donde el array `dim_cm` contiene al menos un elemento cuyo valor es mayor que 23.

```
> db.library.find( { dim_cm: { $gt: 23 } } )
```

Al especificar condiciones compuestas en los array, es posible indicar en la consulta que un solo elemento cumpla con la condición o cualquier combinación de elementos del array cumpla con las condiciones:

```
> db.library.find( { dim_cm: { $gt: 15, $lt: 20 } } )
```

CRUD - Read

```
db.library.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

- ▶ Con ayuda del operador **\$elemMatch** se pueden especificar varios criterios en los elementos de una array de modo que al menos uno de ellos cumpla todos los criterios especificados.

```
> db.library.find( { dim_cm: { $elemMatch: { $gt: 20, $lt: 29 } } } )
```

Utilice la notación de punto para indicar condiciones de consulta sobre un elemento concreto del array. Para ello utilice el índice o posición particular del array. Los arrays usa indexación basada en cero, igual que Java.

- ▶ La consulta también puede hacerse a través del tamaño del array. Para ello, utilice el operador **\$size** para consultar array de un número de elementos concreto.

```
> db.library.find( { "dim_cm.1": { $gt: 25 } } )
```

```
> db.library.find( { "tags": { $size: 3 } } )
```

CRUD - Read

- Compruebe las siguientes consultas:

```
db.library.insertMany( [
  { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ],
  { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
  { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ],
  { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ],
  { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] },
]);
```



```
> db.library.find( { "instock": { warehouse: "A", qty: 5 } } )
```

```
> db.library.find( { "instock": { qty: 5, warehouse: "A" } } )
```

```
> db.library.find( { 'instock.qty': { $lte: 20 } } )
```

```
> db.library.find( { 'instock.0.qty': { $lte: 20 } } )
```

```
> db.library.find( { "instock": { $elemMatch: { qty: 5, warehouse: "A" } } } )
```

```
> db.library.find( { "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } } )
```

```
> db.library.find( { "instock.qty": { $gt: 10, $lte: 20 } } )
```

```
> db.library.find( { "instock.qty": 5, "instock.warehouse": "A" } )
```



CRUD - Read

- Compruebe las siguientes consultas:

```
> db.library.find( { status: "A" } )
```

```
> db.library.find( { status: "A" }, { item: 1, status: 1 } )
```

```
> db.library.find( { status: "A" }, { item: 1, status: 1, _id: 0 } )
```

```
> db.library.find( { status: "A" }, { item: 1, status: 0, _id: 0 } )
```

```
> db.library.find( { status: "A" }, { status: 0, instock: 0 } ) //excluir solo estos
```

```
> db.library.find(  
  { status: "A" },  
  { item: 1, status: 1, "size.uom": 1 }  
)
```

```
db.library.insertMany( [  
  {item: "journal", status: "A", size: { h: 14, w: 21, uom: "cm" }, instock: [{ warehouse: "A", qty: 5 }]},  
  {item: "notebook", status: "A", size: { h: 8.5, w: 11, uom: "in" }, instock: [{ warehouse: "C", qty: 5 }]},  
  {item: "paper", status: "D", size: { h: 8.5, w: 11, uom: "in" }, instock: [{ warehouse: "A", qty: 60 }]},  
  {item: "planner", status: "D", size: { h: 22.85, w: 30, uom: "cm" }, instock: [{ warehouse: "A", qty: 40 }]},  
  {item: "postcard", status: "A", size: { h: 10, w: 15.25, uom: "cm" }, instock: [{ warehouse: "B", qty: 15 },  
    { warehouse: "C", qty: 35 } ] }  
]);
```



CRUD - Read

Proyecciones

- ▶ Permite definir qué campos se incluirán en los resultados.
 - Por defecto, se traen todos los campos.
- ▶ Objeto con campos a incluir o excluir del resultado:
 - La clave indica el nombre campo.
 - Valores a 0: se trae todo menos dichos campos.
 - Valores a 1: sólo se traen esos campos.
 - El `_id` siempre viene, salvo que se excluya explícitamente.
 - No se pueden mezclar valores a 0 y a 1, excepto el `_id`.

```
> db.products.find( )
```

```
> db.products.find( { } )
```

CRUD - Read

Proyecciones

- ▶ Excluir el campo "description":

```
> db.products.find( {}, { "description": 0 } )
```

- ▶ Obtener solo el campo "Price":

```
> db.products.find( {}, { "price": 1, "_id": 0 } )
```

```
> db.products.findOne( {}, { "description": 0 } )
```

```
> db.products.findOne( {}, { "price": 1, "_id": 0 } )
```

CRUD - Read

Ordenación

- ❑ Método asociado al cursor ¿qué significa esto?
- ❑ Por defecto, los datos se obtienen en el orden natural.
 - ❑ Orden en que están almacenados en disco
- ❑ Objeto con campos a utilizar
 - ❑ Es relevante el orden de los campos.
 - ❑ Valor **1** para orden ascendente, **-1** para descendente.

```
> db.products.find().sort( { "price": -1, "name": 1 } )
```

CRUD – Read - Operadores

- ❑ Cadenas que empiezan con **\$** con semántica predefinida.
- ❑ Aplicable a filtros, proyecciones, modificaciones y otros.

Ejemplos

- ❑ Obtener todos los productos que cuestan más de €50

```
> db.products.find({ "price": { "$gt": 50 } })
```

- ❑ Obtener los productos que están en la categoría 8 o que no están activos

```
> db.products.find( { "$or": [ { "category_ids": 8 }, { "is_active": false } ] } )
```

CRUD - Read - Operadores

Comparaciones

\$gt, \$gte, \$lt, \$lte, \$ne, \$in, \$nin

```
> db.products.find( { "primary_category_id": { "$nin": [ 4, 6, 12 ] } } )
```

Lógicos

\$or, \$nor, \$and, \$not

```
> db.products.find( { price: { $not: { $gt: 1.99 } } } )
```

Campos

\$exists, \$type

```
> db.products.find( { "price": { "$type": 2 } } )
```

Consulta la diapositiva
"Tipos de datos"

Tipos de datos (\$type)

A partir de MongoDB 3.2, el operador `$type` acepta alias de cadena para los tipos BSON además de los números correspondientes a los tipos BSON.
Las versiones anteriores solo aceptaban los números correspondientes al tipo BSON.

Importante:
Al comparar valores de diferentes tipos de BSON, MongoDB utiliza el siguiente orden de comparación, de menor a mayor:

Tipo	Número	Alias	Cuidado
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5		
"binData"			
Undefined	6	"undefined"	Deprecated.
Boolean	8	"bool"	
ObjectId	7	"objectId"	
Date	9	"date"	
Regular			
Null	10	"null"	
Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated.
JavaScript	13	"javascript"	
JavaScript (with scope)	14	"javascriptWithScope"	
Symbol	15	"symbol"	
32-bit integer	16	"int"	
Deprecated. scope		"scope"	
Timestamp	17	"timestamp"	3.4.
64-bit integer	18	"long"	New in version
Decimal128	19	"decimal"	
Min key	-1	"minKey"	Tipo interno
Max key	127	"maxKey"	Tipo interno

CRUD - Read - Operadores

Listas

`$all, $elemMatch, $size`

```
> db.products.find( { "category_ids": { "$all": [ 8, 15 ] } } )
```

Avanzados

`$mod, $regex, $where`

```
> db.products.find( { "sku": { "$not": { "$regex": "\w{3}\d{4}" } } } )
```

Índices especiales

`$geoWithin, $geoIntersects, $near, $nearSphere, $text`

```
>db.products.find({"$where":"this.category_ids.indexOf(this.primary_category_id)== -1"})
```


CRUD - Read - Operadores

Comparación

Operador	Descripción
\$eq	Compara valores que son iguales a un valor específico.
\$gt	Compara valores que son mayores que un valor específico.
\$gte	Compara valores que son mayor o igual que un valor específico.
\$in	Coincide con cualquiera de los valores especificados en una matriz.
\$lt	Compara valores que son menores que un valor específico.
\$lte	Compara valores que son menores o igual que un valor específico.
\$ne	Coincide con todos los valores que no son iguales a un valor concreto.
\$nin	No coincide con ninguno de los valores especificados en una matriz.

Lógicos y de Elementos

Operador	Descripción
\$and	Une cláusulas de consulta con un AND lógico y devuelve todos los documentos que coinciden con las condiciones de ambas cláusulas.
\$or	Une las cláusulas de consulta con un OR lógico y devuelve todos los documentos que coinciden con las condiciones de cualquiera de las cláusulas.
\$not	Invierte el efecto de una expresión de consulta y devuelve documentos que no coinciden con la expresión de consulta.
\$nor	Unir las cláusulas de consulta con un NOR lógico devuelve todos los documentos que no coinciden con ambas cláusulas.
\$exists	Coincide con documentos que tienen el campo especificado.
\$type	Selecciona documentos si un campo es del tipo especificado.

CRUD - Read - Operadores

Evaluación

Operador	Descripción
\$expr	Permite el uso de expresiones de agregación dentro del lenguaje de consulta.
\$jsonSchema	Validar documentos contra el esquema JSON dado.
\$mod	Realiza una operación de módulo sobre el valor de un campo y selecciona documentos con un resultado específico.
\$regex	Selecciona documentos donde los valores coinciden con una expresión regular específica.
\$text	Realiza búsqueda de texto.
\$where	Coincide con documentos que satisfacen una expresión de JavaScript.

Array y Proyección

Operador	Descripción
\$all	Coincide con matrices que contienen todos los elementos especificados en la consulta.
\$elemMatch	Selecciona documentos si el elemento en el campo de matriz coincide con todas las condiciones especificadas de \$elemMatch.
\$size	Selecciona documentos si el campo de matriz tiene un tamaño especificado.
\$	Proyecta el primer elemento en una matriz que coincide con la condición de consulta.
\$meta	Proyecta la puntuación del documento asignada durante la operación \$ text.
\$slice	Limita el número de elementos proyectados desde una matriz. Admite saltos y límite.

CRUD - Read - Operadores

Comparación: \$eq

```
{ <field>: { $eq: <value> } }
```

```
{ field: <value> }
```

```
> db.products.find( { price: { $eq: 339.95 } } )
```

```
> db.products.find( { price: 339.95 } )
```

```
> db.products.find( { "variants.1.name": { $eq: "Blue, Small" } } )
```

```
> db.products.find( { "variants.1.name": "Blue, Small" } )
```

```
> db.products.find( { "category_ids.0": { $eq: 7 } } )
```

```
> db.products.find( { "category_ids.0": 7 } )
```

CRUD - Read - Operadores

Comparación: \$gt, \$gte, \$lt, \$lte, \$ne

```
{ <field>: { $eq: <value> } }
```

```
> db.products.find( { price: { $gt: 339.95 } } )
```

```
> db.products.find( { "variants.1.name": { $eq: "Blue, Small" } } )
```

```
> db.products.find( { "category_ids.0": { $gt: 5 } } )
```

CRUD - Read - Operadores

Comparación: \$in, \$nin

```
{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }
```

```
> db.products.find( { price: { $in: [100, 200]} } )
```

```
> db.products.find( { "category_ids.0": { $in: [1, 9] } } )
```

```
> db.products.find( { "name": { $in: [ /^Bu/, /^Be/ ] } } ) // $regex
```

CRUD - Read - Operadores

Lógicos: \$and, \$or

```
{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }
```

```
{ $or: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }
```

```
> db.products.find({ $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )
```

```
> db.products.find({ price: { $ne: 1.99, $exists: true } })
```

```
> db.inventory.find( {  
  $and : [  
    { $or : [ { price : 0.99 }, { price : 1.99 } ] },  
    { $or : [ { is_active : true }, { weight : { $lt : 2 } } ] }  
  ]  
} )
```

CRUD - Read - Operadores

Lógicos: \$not, \$nor

```
{ field: { $not: { <operator-expression> } } }
```

```
{ $nor: [ { <expression1> }, { <expression2> }, ... { <expressionN> } ] }
```

```
> db.products.find( { price: { $not: { $gt: 1.99 } } } )
```

```
> db.products.find( { "name": { $not: { $regex: /^p.* / } } } ) //versión: {$lte: 4.0}
```

```
> db.products.find( { "name": { $not: /^p.* / } } )
```

```
> db.products.find( { $nor: [ { price: 1.99 }, { sale: true } ] } )
```

CRUD - Read - Operadores

Elementos: \$type, \$exists

```
{ field: { $type: <BSON type> } }
```

```
{ field: { $type: [ <BSON type1> , <BSON type2>, ... ] } }
```

```
> db.products.find( { "weight" : { $type : "number" } } )
```

```
> db.products.find( { "name" : { $type : "string" } } )
```

```
> db.products.find( { "category_ids" : { $type : "array" } } ) //versión: {$lte: 3.6}
```


CRUD - Read - Operadores

Evaluación: \$expr

```
{ $expr: {<expression> } } //versión: {$lte: 3.6}
```

```
> db.products.find( { $expr : { $gt : ["$weight", "$price"] } } )
```

```
> db.products.find( { "name" : { $type : "string" } } )
```

CRUD - Read - Operadores

Evaluación: \$mod, \$text

```
{
  $text:
    {
      $search: <string>,

      $language: <string>,
      $caseSensitive: <boolean>,
      $diacriticSensitive: <boolean>
    }
}
```

```
{ field: { $mod: [ divisor, remainder ] } }
```

```
> db.products.find( { weight: { $mod: [ 4, 0 ] } } )
```

```
> db.products.find( { $text: { $search: "bueno" } } )
```

```
> db.products.find( { $text: { $search: "\"buen amigo\"" } } )
```

```
> db.products.find( { $text: { $search: "buen", $language: "es" } } ) //no stopwords
```

```
> db.products.find( { $text: { $search: "buen", $caseSensitive: true } } )
```

CRUD - Read - Operadores

Evaluación: \$where

WARNING

El operador \$where permite pasar una expresión o función JavaScript al sistema de consulta.

\$where proporciona una mayor flexibilidad, pero requiere que la base de datos procese la expresión o función de JavaScript para cada documento de la colección.

Es recomendable que haga referencia al documento en la expresión o función JavaScript utilizando [this](#) u [obj](#)

```
> db.products.find( { $where: function() {  
    return ( this.weight >= 1 )  
} } );
```

WARNING

- ❖ \$expr es más rápido que \$where porque no ejecuta JavaScript. Use \$expr siempre que sea posible.
- ❖ \$where no puede utilizar los índices creados, la consulta va mucho mejor si utiliza \$gt, \$in, etc.
- ❖ Si necesita manipular sus datos con un operador que no exista, recurra a \$where.

WARNING



Instalar librerías

```
npm init -y  
npm install express mongoose ejs body-parser method-override
```

```

const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const methodOverride = require('method-override');
const User = require('./models/User');

const app = express();

// Configuración
app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: true }));
app.use(methodOverride('_method')); // Para soportar PUT y DELETE
app.set('view engine', 'ejs');

// Conexión a MongoDB
mongoose.connect('mongodb://127.0.0.1:27017/usuarios', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

// Rutas
app.get('/', async (req, res) => {
  const users = await User.find();
  res.render('index', { users });
});

```

```

app.get('/users/new', (req, res) => {
  res.render('create');
});

app.post('/users', async (req, res) => {
  const { correo, password, nombre, rol } = req.body;
  await User.create({ correo, password, nombre, rol });
  res.redirect('/');
});

app.get('/users/:id', async (req, res) => {
  const user = await User.findById(req.params.id);
  res.render('show', { user });
});

app.get('/users/:id/edit', async (req, res) => {
  const user = await User.findById(req.params.id);
  res.render('edit', { user });
});

app.put('/users/:id', async (req, res) => {
  const { correo, password, nombre, rol } = req.body;
  await User.findByIdAndUpdate(req.params.id, { correo, password, nombre, rol });
  res.redirect('/');
});

app.delete('/users/:id', async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.redirect('/');
});

app.listen(3000, () => {
  console.log('Servidor corriendo en http://localhost:3000');
});

```

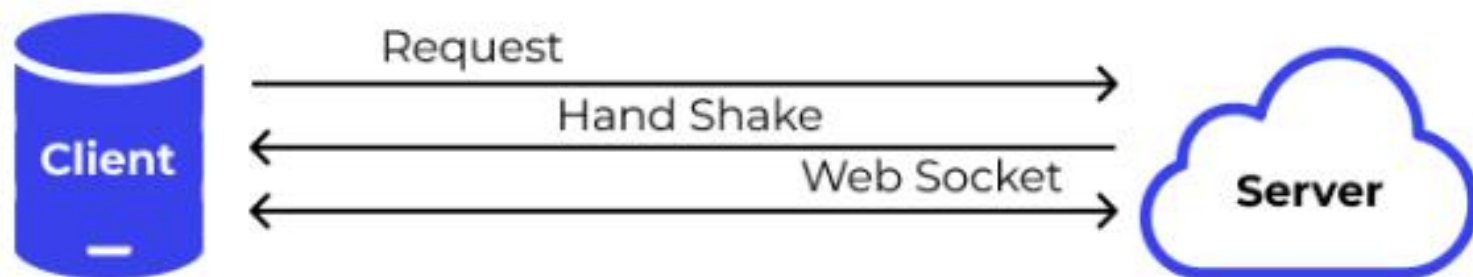
Web Soquets



Qué es WebSocket?

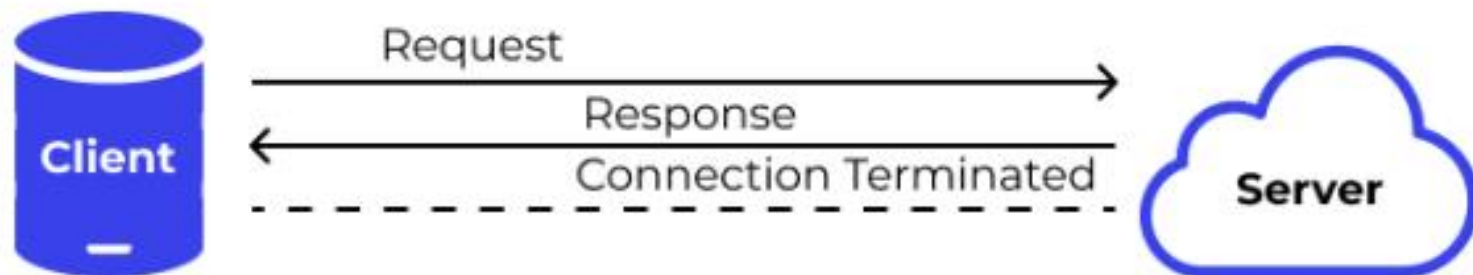
- WebSocket es un protocolo de comunicación que permite la **comunicación bidireccional en tiempo real** entre un cliente (navegador) y un servidor.
- A diferencia de HTTP, donde el cliente debe solicitar información al servidor, con WebSocket **el servidor puede enviar datos al cliente en cualquier momento**, sin necesidad de que el cliente haga una petición.

WebSocket Connection



VS


HTTP Connection





¿Cómo funciona WebSocket en Node.js?

- Node.js es una plataforma ideal para manejar WebSockets porque usa un modelo de eventos no bloqueante, lo que le permite manejar múltiples conexiones simultáneamente de manera eficiente.
- Para usar WebSockets en Node.js, generalmente se usa la librería ws, que facilita la implementación del servidor y la comunicación con los clientes.



Vamos a crear una aplicación sencilla donde un servidor Node.js envía mensajes en tiempo real a todos los clientes conectados.

Ejemplo Practico

1. Instalación de WebSocket en Node.js

En la terminal, crea un nuevo proyecto y añade la librería para manejar WebSocket:

```
mkdir websocket-demo  
cd websocket-demo  
npm init -y  
npm install ws
```

2. Crear el Servidor WebSocket

Crema un archivo server.js y coloca el siguiente código:

```
const WebSocket = require('ws');

// Crear un servidor WebSocket en el puerto 8080
const server = new WebSocket.Server({ port: 8080 });

console.log("Servidor WebSocket escuchando en  
ws://localhost:8080");

server.on("connection", socket => {
  console.log("Cliente conectado");

  // Enviar un mensaje de bienvenida al cliente
  socket.send("Bienvenido al servidor WebSocket");

  // Escuchar mensajes del cliente
  socket.on("message", message => {
    console.log(`Mensaje recibido: ${message}`);

    // Reenviar el mensaje a todos los clientes conectados
    server.clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(`Mensaje de otro cliente: ${message}`);
      }
    });
  });
});
```

```
// Detectar desconexión del cliente
socket.on("close", () => {
  console.log("Cliente desconectado");
});
```

3. Crear el Cliente WebSocket en HTML

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cliente WebSocket</title>
</head>
<body>
  <h1>Cliente WebSocket</h1>
  <input type="text" id="message" placeholder="Escribe un mensaje">
  <button onclick="sendMessage()">Enviar</button>
  <ul id="messages"></ul>

  <script>
    // Conectar al servidor WebSocket
    const socket = new WebSocket("ws://localhost:8080");

    socket.onopen = () => {
      console.log("Conectado al servidor WebSocket");
    };

    // Recibir mensajes del servidor
    socket.onmessage = event => {
      const li = document.createElement("li");
      li.textContent = event.data;
      document.getElementById("messages").appendChild(li);
    };

    // Enviar mensaje al servidor
    function sendMessage() {
      const message = document.getElementById("message").value;
      socket.send(message);
      document.getElementById("message").value = "";
    }
  </script>
</body>
</html>
```



Usos de WebSockets

WebSockets se usan en aplicaciones donde es necesario el **envío y recepción de datos en tiempo real** sin que el cliente tenga que hacer peticiones constantes al servidor.


- Chats y Mensajería Instantánea (WhatsApp, Messenger, Slack, etc.)
- **Caso de uso:** En plataformas de mensajería, los WebSockets permiten enviar y recibir mensajes instantáneamente sin necesidad de recargar la página.
- **Ejemplo:**
 - Un usuario envía un mensaje.
 - El servidor WebSocket lo recibe y lo reenvía al destinatario en tiempo real.
 - El destinatario recibe el mensaje inmediatamente sin hacer una nueva solicitud al servidor.
- **Ejemplo en producción:** WhatsApp Web y Facebook Messenger usan WebSockets para mantener las conversaciones en tiempo real.


- 
- Notificaciones en Tiempo Real (Facebook, Gmail, Twitter, etc.)
 - **Caso de uso:** Se usan WebSockets para enviar notificaciones inmediatas sin que el usuario tenga que actualizar la página.
 - **Ejemplo:**
 - Un usuario recibe un "me gusta" en su foto.
 - El servidor WebSocket envía una notificación al usuario sin que este tenga que actualizar la página.
 - **Ejemplo en producción:** Facebook, Twitter y Gmail muestran notificaciones en tiempo real usando WebSockets.

-
- Juegos Multijugador en Línea (Among Us, Fortnite, etc.) 🎮
 - **Caso de uso:** En juegos en línea, los WebSockets permiten que cada acción del jugador (movimiento, ataque, etc.) se transmita en tiempo real a otros jugadores conectados.

Ejemplo:


- Un jugador dispara en un juego FPS.
- El servidor WebSocket envía la información de disparo a los demás jugadores al instante.
- Los jugadores ven el impacto en tiempo real sin retrasos.
- **Ejemplo en producción:** Juegos como **Fortnite, Agar.io y Among Us** usan WebSockets para sincronizar la información de los jugadores en tiempo real.

-
- Bolsa de Valores y Finanzas en Tiempo Real 
 - **Caso de uso:** Los mercados financieros requieren actualizaciones instantáneas de precios, órdenes y transacciones.
 - **Ejemplo:**
 - Un inversionista está viendo el precio del Bitcoin.
 - El servidor WebSocket envía actualizaciones de precio cada segundo.
 - El inversionista ve los cambios de precio sin necesidad de recargar la página.
 - **Ejemplo en producción:** Plataformas como **Binance**, **eToro** y **TradingView** usan WebSockets para actualizar precios en tiempo real.


- 
- Colaboración en Tiempo Real (Google Docs, Trello, Miro, etc.)
 - **Caso de uso:** Aplicaciones de edición colaborativa necesitan que múltiples usuarios trabajen simultáneamente en un mismo documento o proyecto.

Ejemplo:

- Dos personas editan un documento en Google Docs.
- Los cambios aparecen en tiempo real para ambos sin que tengan que actualizar la página.
- 💡 **Ejemplo en producción: Google Docs, Notion y Figma** usan WebSockets para la colaboración en tiempo real.

- 
- Seguimiento de Transporte y GPS en Tiempo Real (Uber, Cabify, Google Maps, etc.)
 - **Caso de uso:** Aplicaciones de transporte como Uber usan WebSockets para actualizar la ubicación del conductor en tiempo real.
 - **Ejemplo:**
 - Un pasajero solicita un Uber.
 - El servidor WebSocket envía actualizaciones constantes de la ubicación del conductor.
 - El pasajero ve en tiempo real cómo se acerca el auto.
 - **Ejemplo en producción:** Uber, Cabify y Google Maps usan WebSockets para mostrar ubicaciones en tiempo real.

-
- Transmisión en Vivo (YouTube, Twitch, Zoom, etc.) 🎥
 - **Caso de uso:** Los WebSockets permiten enviar comentarios y reacciones en tiempo real durante una transmisión en vivo.
 - **Ejemplo:**
 - Un streamer está en vivo en Twitch.
 - Los espectadores envían mensajes y emojis en el chat.
 - Todos los mensajes aparecen al instante sin necesidad de recargar la página.
 - **Ejemplo en producción:** YouTube Live, Twitch y Zoom usan WebSockets para chats y reacciones en vivo

- 
- IoT (Internet de las Cosas) y Domótica (Smart Homes, Sensores, etc.)
 - **Caso de uso:** WebSockets permiten la comunicación en tiempo real entre dispositivos IoT y el usuario.
 - **Ejemplo:**
 - Un usuario enciende una luz desde su celular.
 - El servidor WebSocket envía la señal al sistema de domótica.
 - La luz se enciende inmediatamente.
 - **Ejemplo en producción:** Google Nest, Philips Hue y Amazon Echo usan WebSockets para controlar dispositivos inteligentes.