

Desarrollo Web en Entorno Cliente

ÍNDICE DE CONTENIDO

- 1. ¿Qué es JSON Server?**
- 2. Instalación**
 - 2.1. Requisitos previos**
 - 2.2. Comando de instalación**
 - 2.3. Estructura básica del proyecto**
- 3. Configuración**
 - 3.1. Creación del archivo db.json**
 - 3.2. Estructura de datos**
 - 3.3. Ejemplo de configuración**
- 4. Uso básico**
 - 4.1. Comandos para iniciar el servidor**
 - 4.2. Acceso a recursos y endpoints REST**
 - 4.3. Ejemplo con fetch() en JavaScript**
- 5. Operaciones CRUD**
 - 5.1. GET, POST, PUT, PATCH y DELETE**
 - 5.2. Ejemplos prácticos con código**
- 6. Rutas avanzadas y filtros**
 - 6.1. Parámetros de consulta (_sort, _order, _limit, etc.)**
 - 6.2. Ejemplo de filtrado y ordenación**
- 7. AJAX moderno: usando "fetch" en lugar del objeto XMLHttpRequest**

Desarrollo Web en Entorno Cliente

1.- ¿QUÉ ES JSON SERVER?

JSON Server es una herramienta que permite crear de forma rápida y sencilla una **API REST falsa (mock API)** a partir de un archivo JSON.

Su función principal es simular un **servidor backend** sin necesidad de programar uno real, utilizando un archivo que actúa como base de datos (normalmente llamado db.json).

JSON Server está construido sobre **Node.js** y utiliza el formato **JSON (JavaScript Object Notation)** para almacenar y servir los datos. Cada colección de datos dentro del archivo JSON se convierte automáticamente en un **endpoint REST** accesible mediante peticiones HTTP estándar (GET, POST, PUT, PATCH y DELETE).

En otras palabras, JSON Server convierte un simple archivo JSON en una **API completamente funcional**, ideal para pruebas, desarrollo y aprendizaje.

Para qué se utiliza en el desarrollo web

En el desarrollo web, especialmente durante la creación del **frontend**, muchas veces se necesita acceder a datos que aún no están disponibles porque el **backend real todavía no está implementado**.

En esos casos, JSON Server se utiliza como una **herramienta de simulación** para:

Probar interfaces que dependen de datos externos.

Desarrollar aplicaciones SPA (Single Page Application) sin necesidad de tener un servidor real.

Realizar peticiones AJAX o fetch() desde JavaScript y comprobar cómo responde una API.

Practicar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sobre datos de ejemplo.

De esta forma, los desarrolladores pueden trabajar de forma **independiente y paralela**: el equipo de frontend avanza con la interfaz mientras el backend se desarrolla por separado.

Ventajas frente a un backend real durante el desarrollo

El uso de JSON Server ofrece numerosas ventajas frente a la utilización de un servidor backend real en fases tempranas del desarrollo:

1. **Rapidez de configuración**: solo se necesita instalar la herramienta y crear un archivo JSON. No hay que programar lógica de servidor ni configurar bases de datos complejas.
2. **Ligereza y simplicidad**: al no requerir frameworks pesados ni dependencias complejas, se ejecuta en segundos.

Desarrollo Web en Entorno Cliente

3. **Ideal para pruebas y aprendizaje:** permite simular situaciones reales de trabajo con APIs REST, lo que lo hace perfecto para estudiantes o entornos de prácticas.
4. **Desarrollo paralelo:** los equipos de frontend y backend pueden trabajar simultáneamente sin depender uno del otro.
5. **Reinicio rápido de datos:** al modificar el archivo db.json, los cambios se reflejan de inmediato, sin necesidad de reiniciar servicios ni compilar código.
6. **Total compatibilidad con herramientas modernas:** funciona perfectamente con frameworks como **React, Vue, Angular** o con aplicaciones JavaScript puras.

En resumen, JSON Server es una herramienta sencilla pero muy potente para simular un entorno backend, acelerar el desarrollo y facilitar la práctica con APIs RESTful.

2.- INSTALACIÓN

2.1. Requisitos previos

Para poder utilizar **JSON Server**, es necesario tener instalado **Node.js** en el equipo. Node.js es un entorno de ejecución que permite ejecutar código JavaScript fuera del navegador, lo que facilita la creación de servidores y herramientas de desarrollo.

Cuando se instala Node.js, también se instala automáticamente **npm (Node Package Manager)**, el gestor de paquetes de Node, que permite descargar e instalar librerías y dependencias, como JSON Server.

Se puede descargar el instalador desde la página oficial: <https://nodejs.org/es>

Pasos para verificar la instalación:

1. Abre una terminal o consola de comandos.
2. Escribe los siguientes comandos para comprobar si Node.js y npm están instalados:

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

2.2. Comando de instalación

Una vez instalado Node.js, la instalación de **JSON Server** es muy sencilla. Puede hacerse de dos maneras:

- **Instalación global:** Permite ejecutar JSON Server desde cualquier ubicación del sistema

```
npm install -g json-server
```

Desarrollo Web en Entorno Cliente

- **Instalación local:** Se instala únicamente dentro de un proyecto determinado. Es la opción más común en entornos de desarrollo

```
npm install json-server --save-dev
```

Después de la instalación, se puede comprobar si JSON Server está disponible ejecutando:

```
json-server --version
```

2.3. Estructura básica del proyecto

Una vez instalado JSON Server, el siguiente paso es crear la **estructura mínima de archivos** para que el servidor funcione. Un ejemplo básico de estructura sería:

- **db.json** => Es el archivo principal de datos. Actúa como una pequeña “base de datos” en formato JSON. Ejemplo de contenido:

```
{
  "usuarios": [
    { "id": 1, "nombre": "Ana", "email": "ana@example.com" },
    { "id": 2, "nombre": "Carlos", "email": "carlos@example.com" }
  ],
  "tareas": [
    { "id": 1, "titulo": "Estudiar JSON Server", "completada": false }
  ]
}
```

- **package.json** → Archivo de configuración de Node.js donde se almacenan las dependencias del proyecto (como JSON Server) y scripts personalizados.
- **Package-lock.json** → El archivo **registra las versiones exactas** de todas las dependencias (y sus dependencias) que se instalan en el proyecto.
Esto significa que:
 - Si alguien más clona tu proyecto y ejecuta npm install, tendrá **exactamente las mismas versiones** de los paquetes que tú usaste.
 - Evita inconsistencias entre entornos (por ejemplo, tu equipo local y el servidor de producción).

3.- CONFIGURACIÓN

3.1. Creación del archivo db.json

El archivo **db.json** es el núcleo de la configuración del servidor simulado. Para crearlo, sigue estos pasos:

1. En la raíz del proyecto, crea un archivo llamado **db.json**.

Desarrollo Web en Entorno Cliente

- Este archivo contendrá la información que actuará como base de datos en formato **JSON** (JavaScript Object Notation).

3.2. Estructura de datos

El archivo **db.json** está organizado en **colecciones**, que representan entidades o tablas dentro de la base de datos.

Cada colección contiene un **array de objetos**, donde cada objeto representa un registro.

Ejemplo general de estructura:

```
{  
  "usuarios": [  
    { "id": 1, "nombre": "Carlos López", "email": "carlos@example.com", "activo": true },  
    { "id": 2, "nombre": "Marta Díaz", "email": "marta@example.com", "activo": false }  
  ]  
}
```

Cada configuración se adapta según las necesidades del proyecto, pero la estructura general del archivo db.json permanece igual: un conjunto de colecciones con registros en formato JSON.

Por defecto, **JSON Server** busca un archivo llamado db.json, pero **puedes usar cualquier nombre** siempre que se lo indiques al ejecutar el servidor.

Productos.json

```
{  
  "productos": [  
    { "id": 1,"categoriaid": 1,"nombre": "Teclado mecánico", "precio": 49.99, "stock": 25 },  
    { "id": 2, "categoriaid": 1,"nombre": "Ratón inalámbrico", "precio": 29.95, "stock": 40 }  
  ],  
  "categorias": [  
    { "id": 1, "nombre": "Periféricos" },  
    { "id": 2, "nombre": "Componentes" }  
  ]  
}
```

Para iniciar el servidor con los datos del archivo db.json, se puede ejecutar el siguiente comando:

```
json-server db.json  
http://localhost:3000/usuarios  
http://localhost:3000/tareas
```

Desarrollo Web en Entorno Cliente

4.- OPERACIONES CRUD

4.1. GET, POST, PUT, PATCH y DELETE

El término **CRUD** se refiere a las **cuatro operaciones básicas** que puede realizar una aplicación sobre una base de datos o una API:

Create (crear), Read (leer), Update (actualizar) y Delete (eliminar).

En el caso de **JSON Server**, estas operaciones se implementan mediante los **métodos HTTP**: POST, GET, PUT, PATCH y DELETE.

Estas acciones permiten simular completamente el funcionamiento de una API REST, ideal para desarrollo y pruebas.

MÉTODO	ACCIÓN	DESCRIPCIÓN	EJEMPLO
GET	Leer	Obtiene datos del servidor. Puede usarse para listar todos los registros o uno específico.	`/usuarios` `/usuarios/1`
POST	Crear	Envía datos al servidor para crear un nuevo registro.	`/usuarios`
PUT	Actualizar (completo)	Reemplaza totalmente un registro existente por uno nuevo.	`/usuarios/1`
PATCH	Actualizar (parcial)	Modifica solo algunos campos de un registro.	`/usuarios/1`
DELETE	Eliminar	Borra un registro existente.	`/usuarios/1`

4.2. ejemplos prácticos con código

Archivo base (db.json)

```
{  
  "usuarios": [  
    { "id": 1, "nombre": "Carlos López", "email": "carlos@example.com", "activo": true },  
    { "id": 2, "nombre": "Marta Díaz", "email": "marta@example.com", "activo": false }  
  ]  
}
```

Desarrollo Web en Entorno Cliente

A) Obtener datos (GET)

```
// Obtener todos los usuarios
fetch('http://localhost:3000/usuarios')
.then(response => response.json())
.then(data => {
  console.log('Usuarios:', data);
})
.catch(error => console.error('Error al obtener usuarios:', error));

// Obtener un usuario específico por ID
fetch('http://localhost:3000/usuarios/1')
.then(response => response.json())
.then(usuario => console.log('Usuario con ID 1:', usuario))
.catch(error => console.error('Error al obtener usuario:', error));
```

B) Crear un nuevo registro (POST)

```
fetch('http://localhost:3000/usuarios', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    nombre: 'María López',
    email: 'maria@example.com'
  })
})
.then(response => response.json())
.then(data => console.log('Usuario creado:', data))
.catch(error => console.error('Error al crear usuario:', error));
```

C) Actualizar un registro completo (PUT)

```
fetch('http://localhost:3000/usuarios/2', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    id: 2,
    nombre: 'Luis Pérez Actualizado',
    email: 'luis.actualizado@example.com'
  })
})
.then(response => response.json())
.then(data => console.log('Usuario actualizado completamente:', data))
.catch(error => console.error('Error al actualizar usuario:', error));
```

Desarrollo Web en Entorno Cliente

D) Actualizar parcialmente un registro (PATCH)

```
fetch('http://localhost:3000/usuarios/1', {
  method: 'PATCH',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    email: 'ana.actualizado@example.com'
  })
})
.then(response => response.json())
.then(data => console.log('Usuario modificado parcialmente:', data))
.catch(error => console.error('Error al modificar usuario:', error));
```

E) Eliminar un registro (DELETE)

```
fetch('http://localhost:3000/usuarios/1', {
  method: 'DELETE'
})
.then(() => console.log('Usuario eliminado correctamente'))
.catch(error => console.error('Error al eliminar usuario:', error));
```

4.3. Ejemplo completo

Index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Gestión de Usuarios (CRUD con JSON Server)</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 40px;
      background-color: #f7f9fc;
      color: #333;
    }
    h1 {
      text-align: center;
      color: #2c3e50;
    }
    form {
      background: #fff;
      padding: 20px;
      border-radius: 10px;
    }
  </style>
</head>
<body>
  <h1>Gestión de Usuarios</h1>
  <form>
    <input type="text" placeholder="Nombre" />
    <input type="text" placeholder="Apellido" />
    <input type="text" placeholder="Email" />
    <input type="password" placeholder="Contraseña" />
    <input type="button" value="Registrar" />
  </form>
</body>
</html>
```

Desarrollo Web en Entorno Cliente

```
        box-shadow: 0 2px 5px rgba(0,0,0,0.1);
        margin-bottom: 30px;
    }
    input, button {
        padding: 10px;
        margin: 5px;
    }
    table {
        width: 100%;
        border-collapse: collapse;
        background: #fff;
    }
    th, td {
        padding: 10px;
        text-align: left;
        border-bottom: 1px solid #ddd;
    }
    tr:hover {
        background-color: #f1f1f1;
    }
    button {
        cursor: pointer;
        border: none;
        border-radius: 5px;
    }
    .btn-editar {
        background-color: #3498db;
        color: white;
    }
    .btn-eliminar {
        background-color: #e74c3c;
        color: white;
    }
</style>
</head>
<body>
    <h1>Gestión de Usuarios</h1>

    <!-- Formulario para agregar o editar usuarios -->
    <form id="usuarioForm">
        <input type="hidden" id="usuarioid">
        <input type="text" id="nombre" placeholder="Nombre" required>
        <input type="email" id="email" placeholder="Correo electrónico" required>
        <label>
            <input type="checkbox" id="activo"> Activo
        </label>
        <button type="submit">Guardar</button>
```

Desarrollo Web en Entorno Cliente

```
</form>

<!-- Tabla para mostrar los usuarios -->
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Nombre</th>
      <th>Email</th>
      <th>Activo</th>
      <th>Acciones</th>
    </tr>
  </thead>
  <tbody id="tablaUsuarios"></tbody>
</table>

<script>
const API_URL = 'http://localhost:3000/usuarios';

const tablaUsuarios = document.getElementById('tablaUsuarios');
const form = document.getElementById('usuarioForm');
const idInput = document.getElementById('usuarioid');
const nombreInput = document.getElementById('nombre');
const emailInput = document.getElementById('email');
const activoInput = document.getElementById('activo');

// Leer (GET)
async function cargarUsuarios() {
  const respuesta = await fetch(API_URL);
  const usuarios = await respuesta.json();

  tablaUsuarios.innerHTML = '';
  usuarios.forEach(usuario => {
    const fila = document.createElement('tr');
    fila.innerHTML = `
      <td>${usuario.id}</td>
      <td>${usuario.nombre}</td>
      <td>${usuario.email}</td>
      <td>${usuario.activo ? 'Si' : 'No'}</td>
      <td>
        <button class="btn-editar" onclick="editarUsuario(${usuario.id})">Editar</button>
        <button class="btn-eliminar" onclick="eliminarUsuario(${usuario.id})">Eliminar</button>
      </td>
    `;
  });
}

cargarUsuarios();
```

Desarrollo Web en Entorno Cliente

```
    tablaUsuarios.appendChild(fila);
  });

}

// Crear o Actualizar
form.addEventListener('submit', async (e) => {
  e.preventDefault();
  const id = idInput.value;
  const nombre = nombreInput.value.trim();
  const email = emailInput.value.trim();
  const activo = activoInput.checked;

  const usuario = { nombre, email, activo };

  if (id) {
    // Actualizar (PUT)
    await fetch(`${API_URL}/${id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ id: Number(id), ...usuario })
    });
  } else {
    // Crear (POST)
    await fetch(API_URL, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(usuario)
    });
  }

  form.reset();
  idInput.value = "";
  cargarUsuarios();
});

// Editar (Carga los datos en el formulario)
async function editarUsuario(id) {
  const respuesta = await fetch(`${API_URL}/${id}`);
  const usuario = await respuesta.json();

  idInput.value = usuario.id;
  nombreInput.value = usuario.nombre;
  emailInput.value = usuario.email;
  activoInput.checked = usuario.activo;
}

// Eliminar (DELETE)
```

Desarrollo Web en Entorno Cliente

```
async function eliminarUsuario(id) {  
  if (confirm('¿Seguro que deseas eliminar este usuario?')) {  
    await fetch(`.${API_URL}/${id}`, { method: 'DELETE' });  
    cargarUsuarios();  
  }  
}  
  
// Cargar los usuarios al iniciar  
cargarUsuarios();  
</script>  
</body>  
</html>
```

5.- RUTAS AVANZADAS Y FILTROS

5.1. Parámetros de consulta (_sort, _order, -limit, etc.)

Además de las operaciones básicas CRUD, **JSON Server** permite realizar consultas más específicas a través de **parámetros en la URL**, muy similares a los que ofrecen las APIs REST profesionales.

Estas opciones permiten **filtrar, ordenar, limitar o combinar datos** sin necesidad de escribir código adicional del lado del servidor.

A) Filtrado por campos:

Permite obtener solo los registros que cumplan con una condición específica.

Ejemplo: GET <http://localhost:3000/usuarios?activo=true>

Devuelve únicamente los usuarios cuyo campo activo es true. También se pueden combinar varios filtros:

GET <http://localhost:3000/usuarios?activo=true&nombre=Marta%20Díaz>

B) Búsquedas parciales

Permite buscar un texto dentro de todos los campos del recurso.

Ejemplo: GET <http://localhost:3000/usuarios?q=carlos>

Devuelve todos los registros que contengan “carlos” en cualquier campo (nombre, email, etc.).

Desarrollo Web en Entorno Cliente

C) Ordenación

Permite ordenar los resultados por uno o más campos.

Ejemplo: GET `http://localhost:3000/usuarios?_sort=activo,nombre&_order=desc,asc`

Devuelve los usuarios ordenados si esta activo y por nombre en orden que indicamos.

D) Límites y paginación

`_limit`: define el número máximo de registros a mostrar.

`_page`: define la página a obtener (usado junto con `_limit`).

Ejemplo: GET `http://localhost:3000/usuarios?_limit=2&_page=1`

Muestra los primeros 2 registros de la primera página.

E) Rangos

Permiten filtrar por valores numéricos o booleanos.

PARÁMETRO	SIGNIFICADO	EJEMPLO
<code>_gte</code>	Mayor o igual que	<code>/productos?precios_gte=20</code>
<code>_lte</code>	Menor o igual que	<code>/productos?precios_lte=20</code>
<code>_ne</code>	Distinto de	<code>/productos?precios_ne=true</code>

F) Referencias y relaciones

Permiten incluir datos de colecciones relacionadas.

Ejemplo: GET `/usuarios?_embed=posts`

Devuelve los usuarios junto con sus publicaciones.

E) Eliminar un registro (DELETE)

Permite **eliminar un recurso** (registro) de la base de datos simulada

Ejemplo: DELETE `/usuarios/2`

Elimina el registro con el id 2

Desarrollo Web en Entorno Cliente

5.2. Ejemplo completo

Archivo base db.json

```
{  
  "usuarios": [  
    { "id": 1, "nombre": "Carlos López", "email": "carlos@example.com", "activo": true },  
    { "id": 2, "nombre": "Marta Díaz", "email": "marta@example.com", "activo": false },  
    { "id": 3, "nombre": "Ana Gómez", "email": "ana@example.com", "activo": true },  
    { "id": 4, "nombre": "Luis Pérez", "email": "luis@example.com", "activo": false }  
  ]  
}
```

Index.html

```
<!DOCTYPE html>  
<html lang="es">  
<head>  
  <meta charset="UTF-8">  
  <title>Filtros y Ordenación - JSON Server</title>  
  <style>  
    body {  
      font-family: Arial, sans-serif;  
      margin: 40px;  
      background-color: #f7f9fc;  
      color: #333;  
    }  
    h1 {  
      text-align: center;  
      color: #2c3e50;  
    }  
    .filtros {  
      display: flex;  
      gap: 10px;  
      justify-content: center;  
      margin-bottom: 25px;  
      flex-wrap: wrap;  
    }  
    input, select, button {  
      padding: 8px;  
      border-radius: 5px;  
      border: 1px solid #ccc;  
    }  
    table {  
      width: 100%;  
      background: #fff;  
      border-collapse: collapse;  
      box-shadow: 0 2px 5px rgba(0,0,0,0.1);  
    }</style>
```

Desarrollo Web en Entorno Cliente

```
}

th, td {
  padding: 10px;
  text-align: left;
  border-bottom: 1px solid #ddd;
}

th {
  background-color: #e3e8f0;
}

tr:hover {
  background-color: #f1f1f1;
}

</style>
</head>
<body>
  <h1>Filtrado y Ordenación de Usuarios</h1>

  <div class="filtros">
    <input type="text" id="busqueda" placeholder="Buscar por nombre...">
    <select id="estado">
      <option value="">Todos</option>
      <option value="true">Activos</option>
      <option value="false">Inactivos</option>
    </select>
    <select id="orden">
      <option value="asc">A-Z</option>
      <option value="desc">Z-A</option>
    </select>
    <button onclick="aplicarFiltros()">Filtrar</button>
    <button onclick="resetFiltros()">Restablecer</button>
  </div>

  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Nombre</th>
        <th>Email</th>
        <th>Activo</th>
      </tr>
    </thead>
    <tbody id="tablaUsuarios"></tbody>
  </table>

  <script>
    const API_URL = 'http://localhost:3000/usuarios';
    const tabla = document.getElementById('tablaUsuarios');
```

Desarrollo Web en Entorno Cliente

```
const busqueda = document.getElementById('busqueda');
const estado = document.getElementById('estado');
const orden = document.getElementById('orden');

// Función principal: carga los usuarios según los filtros aplicados
async function aplicarFiltros() {
  let url = `${API_URL}?_sort=nombre&_order=${orden.value}`;

  // Si hay texto en la búsqueda
  if (busqueda.value.trim() !== "") {
    url += `&q=${encodeURIComponent(busqueda.value.trim())}`;
  }

  // Si se selecciona estado activo/inactivo
  if (estado.value !== "") {
    url += `&activo=${estado.value}`;
  }

  const respuesta = await fetch(url);
  const usuarios = await respuesta.json();
  mostrarUsuarios(usuarios);
}

// Mostrar los resultados en la tabla
function mostrarUsuarios(usuarios) {
  tabla.innerHTML = "";

  if (usuarios.length === 0) {
    tabla.innerHTML = `|  |  |  |  |
| --- | --- | --- | --- |
| No se encontraron resultados | | | |
`;
    return;
  }

  usuarios.forEach(u => {
    const fila = document.createElement('tr');
    fila.innerHTML = `
      <td>${u.id}</td>
      <td>${u.nombre}</td>
      <td>${u.email}</td>
      <td>${u.activo ? 'Si' : 'No'}</td>
    `;
    tabla.appendChild(fila);
  });
}

// Restablecer filtros
```

Desarrollo Web en Entorno Cliente

```
function resetFiltros() {  
    busqueda.value = "";  
    estado.value = "";  
    orden.value = 'asc';  
    aplicarFiltros();  
}  
  
// Cargar datos iniciales al abrir la página  
aplicarFiltros();  
</script>  
</body>  
</html>
```