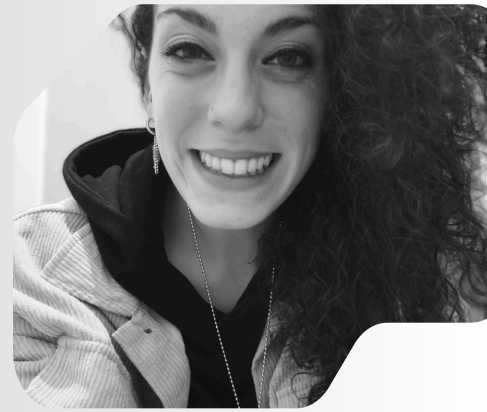


Verificación y actualización de contenido en una aplicación Web

 Desarrollo Web Entorno Servidor



Fecha: Apr 22, 2025

Realizado por: Lidia García Muñoz



https://github.com/Lidiagarmu/reserva_restaurante_app

**se puede acceder al código fuente de la aplicación en el enlace*



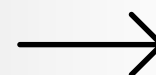
www.linkedin.com/in/lidia-garcia-muñoz

CONTENIDO

1. Introducción	2
2. Tecnologías y herramientas utilizadas	4
3. Arquitectura del proyecto: patrón MVC	5
4. Validación y retroalimentación de formularios	12
◦ 4.1 Validación en tiempo real	
◦ 4.2 Retroalimentación visual al usuario	
◦ 4.3 Vista <i>Admin</i>	
5. Modificación dinámica de formularios	20
◦ 5.1 AJAX + Fetch	
◦ 5.2 WebSockets	
6. Conclusiones	21
7. Fuentes externas y recursos utilizados	22



1 - Introducción



En esta presentación vamos a explorar el desarrollo de una aplicación web diseñada específicamente para la **gestión de reservas de mesa** en un restaurante.

Esta herramienta no solo facilita a los clientes la reserva de manera rápida y sencilla desde cualquier dispositivo, sino que también **incluye una vista informativa de las reservas**, pensada para el personal del restaurante. Desde esta vista admin, se pueden gestionar reservas, verificar disponibilidad en tiempo real y actualizar el contenido dinámicamente, asegurando una experiencia fluida y actualizada tanto para los usuarios como para los administradores.

Uno de los **aspectos clave de esta aplicación es su interactividad**: los usuarios reciben confirmaciones inmediatas, y los administradores pueden responder o modificar reservas al

instante. Todo esto se logra mediante el uso de tecnologías modernas que permiten un desarrollo ágil, eficiente y escalable.

Veamos en este documento, cómo implementar todo esto.

App “Reserva de mesa” vista index

Reservar una Mesa

Nombre completo: *

Tu nombre completo

Fecha: *

Selecciona una fecha

Hora: *

Selecciona una hora

Personas: *

Número de personas

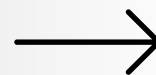
Reservar

App “Reserva de mesa” vista admin

Lista de Reservas

Nombre completo	ID	Fecha	Hora	Personas
Fran Costa	26	11/06/2025	15:00:00	4
Sergio Sánchez	28	25/07/2025	15:00:00	2

2 - Tecnologías y herramientas utilizadas



Esta aplicación ha sido desarrollada utilizando **tecnologías** tanto del **lado del cliente** (**frontend**), donde cierta parte de su funcionalidad ocurre directamente en el navegador del usuario, así como del **lado servidor** (**backend**). Este último, llevará a cabo la parte lógica de la *app* y hará de intermediario entre las vistas y el modelo e interactuará con la base de datos **MySQL** para la obtención de datos y posterior envío al lado cliente.



HTML

Lenguaje de marcado estándar utilizado **para estructurar y presentar contenido en la web** a través de un conjunto de etiquetas y atributos.



Bootstrap

Framework de CSS que **facilita la creación de sitios web responsivos y modernos** con un sistema de grid, componentes pre-diseñados y plugins de JavaScript.



Javascript

Lenguaje de programación que **añade interactividad**, permitiendo que los usuarios interactúen dinámicamente con los elementos de la página.



CSS

Lenguaje utilizado para describir la **aparición y el diseño de una página web**. Se utiliza junto con HTML para controlar cómo se ven los elementos en la pantalla.



Node.js es un **entorno que permite ejecutar JavaScript en el servidor**, usado para crear aplicaciones web rápidas y escalables.



Express es un *framework* para Node.js que **facilita la creación de rutas, servidores y APIs** de forma organizada y eficiente.

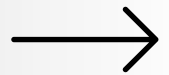


EJS es un **motor de plantillas que permite insertar datos dinámicos en archivos HTML** usando JavaScript en el servidor.



MySQL es un **sistema de base de datos relacional** que guarda, organiza y consulta información estructurada mediante SQL.

3 - Arquitectura del proyecto: patrón MVC



El patrón **MVC (Modelo - Vista - Controlador)** es una forma de estructurar aplicaciones dividiendo su lógica en tres partes principales:

- **Modelo (Model):** gestiona los datos y la lógica de negocio.
- **Vista (View):** se encarga de la interfaz de usuario (lo que el usuario ve).
- **Controlador (Controller):** actúa como intermediario entre el modelo y la vista, procesando las acciones del usuario y actualizando la vista o el modelo según corresponda.

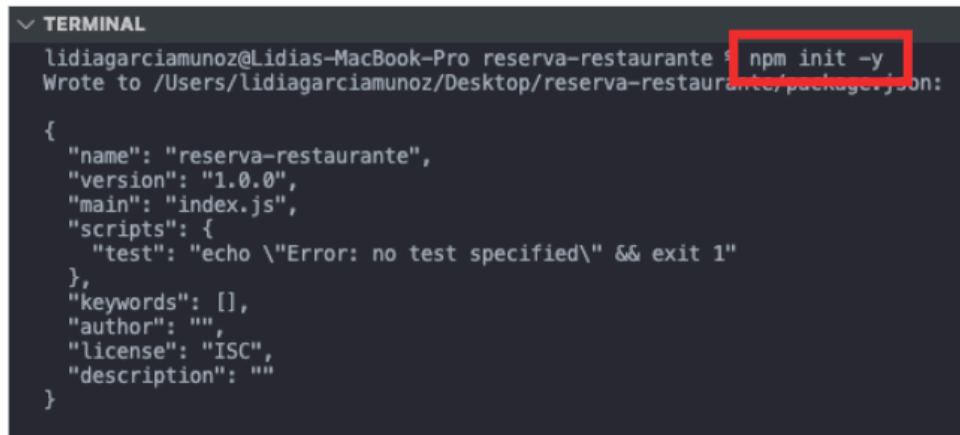
Esta separación facilita el mantenimiento, escalabilidad y organización del código.

¿Cómo iniciar el proyecto?

Lo primero que debemos hacer es crear la carpeta de proyecto. En este caso la llamaremos “reserva-restaurant”.

Para esta aplicación vamos a trabajar con el entorno de desarrollo creado por Microsoft, **Visual Studio Code**. A través de la terminal de este IDE seguiremos los siguientes pasos:

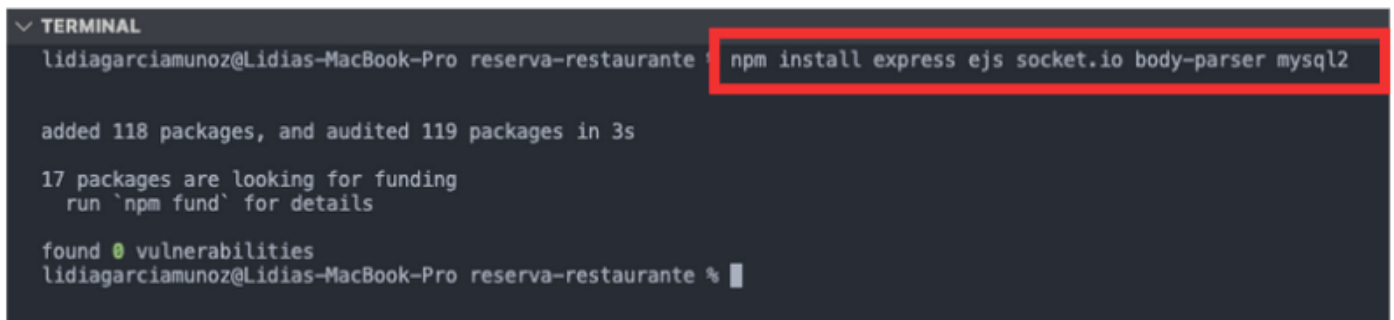
1) Crear proyecto con el comando: *npm init -y*



```
lidiagarciamunoz@Lidias-MacBook-Pro reserva-restaurant: $ npm init -y
Wrote to /Users/lidiagarciamunoz/Desktop/reserva-restaurant/package.json:

{
  "name": "reserva-restaurant",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

2) Instalación de las dependencias del proyecto



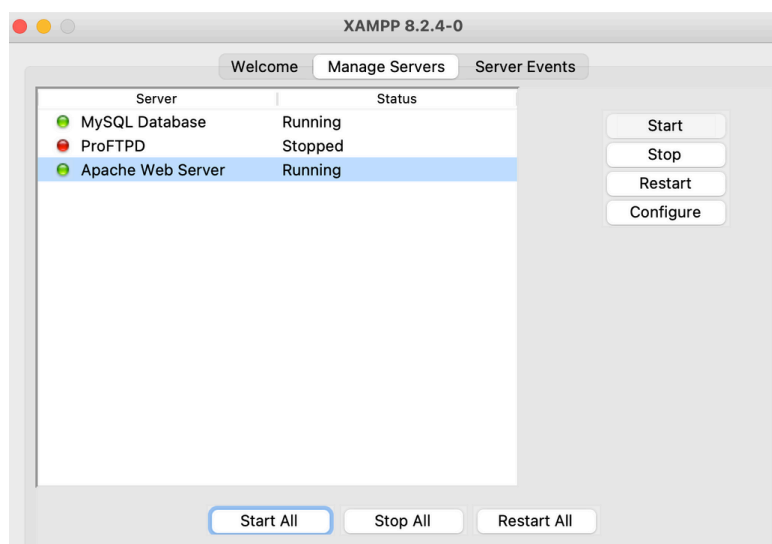
```
lidiagarciamunoz@Lidias-MacBook-Pro reserva-restaurant: $ npm install express ejs socket.io body-parser mysql2

added 118 packages, and audited 119 packages in 3s

17 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
lidiagarciamunoz@Lidias-MacBook-Pro reserva-restaurant: %
```

3) Iniciamos el servidor *Apache Server* a través de *XAMPP*



4) Creamos la base de datos “*reservas_restaurante*” y su correspondiente tabla (insertamos datos en las filas opcionalmente)

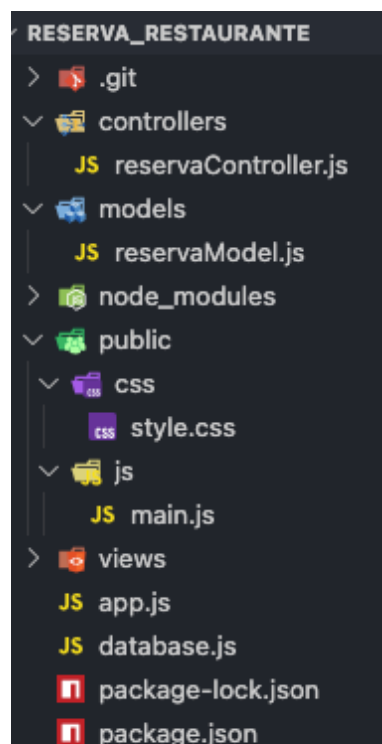


id	fecha	hora	nombre_completo	personas
28	2025-07-25	15:00:00	Sergio Sánchez	2
29	2025-07-23	15:00:00	Lidia García	1
26	2025-06-11	15:00:00	Fran Costa	4
30	2025-05-24	21:00:00	Roberto Muñoz	4

¡Ya tenemos todo listo para comenzar a crear la estructura del proyecto!

El proyecto estará organizado siguiendo este **patrón MVC**. Vamos a explicar cada parte:

Estructura del proyecto



Parte Backend (Lógica del servidor)

- *models/reservaModel.js* – Modelo

- Aquí se define la estructura de los datos y la lógica de negocio relacionada con las reservas.
- Por ejemplo, guardar, leer o eliminar reservas de la base de datos.

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

const Reserva = {
  verificarDisponibilidad: (fecha, hora, callback) => {
    db.query('SELECT * FROM reservas WHERE fecha = ? AND hora = ?', [fecha, hora], callback);
  },
  crearReserva: (datos, callback) => {
    db.query('INSERT INTO reservas (fecha, hora, personas, nombre_completo) VALUES (?, ?, ?, ?)', datos, callback);
  },
  obtenerReservas: (callback) => {
    db.query('SELECT * FROM reservas ORDER BY fecha, hora', callback);
  }
};
```

● **controllers/reservaController.js – Controlador**

- Recibe las peticiones del usuario (como hacer una reserva).
- Llama al modelo si necesita manipular datos y luego muestra la vista correspondiente.
- Controla rutas como /, /reservar, y /admin.

```
const Reserva = require('../models/reservaModel');

exports.formularioReserva = (req, res) => {
  res.render('index');
};

exports.hacerReserva = (req, res) => {
  const { fecha, hora, personas, nombre_completo } = req.body;

  Reserva.verificarDisponibilidad(fecha, hora, (err, resultados) => {
    if (resultados.length > 0) {
      return res.send('Esta fecha y hora ya están reservadas.');
    }
  });

  Reserva.crearReserva([fecha, hora, personas, nombre_completo], (err) => {
    if (err) throw err;
    const io = req.app.get('io');
    io.emit('nuevaReserva'); // Avisamos a todos los usuarios
    res.redirect('/');
  });
});

exports.verReservas = (req, res) => {
  Reserva.obtenerReservas((err, reservas) => {
    res.render('admin', { reservas });
  });
};
```

● **app.js – Archivo principal del backend**

- Configura el servidor Express.
- Define *middlewares* como *body-parser*, rutas, vistas, y *WebSockets* (*socket.io*).

- Es el núcleo de la aplicación, donde se inicializa todo el backend.

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');    503.7k (gzipped: 214.6k)
const socket = require('socket.io');    327.8k (gzipped: 69.5k)
const path = require('path');
const reservaController = require('./controllers/reservaController');

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

const io = socket(server);

// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static('public'));
app.set('view engine', 'ejs');

// Rutas
app.get('/', reservaController.formularioReserva);
app.post('/reservar', reservaController.hacerReserva);
app.get('/admin', reservaController.verReservas);

// WebSockets
io.on('connection', (socket) => {
  console.log('Nueva conexión WebSocket');
  socket.on('nuevaReserva', () => {
    io.emit('actualizarReservas');
  });
});

app.set('io', io); // Para poder usarlo en el controller
```

- **database.js**

- Se encarga de la conexión a la base de datos.
- También forma parte del backend.

```
const mysql = require('mysql2');    791.3k (gzipped: 347.4k)

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'reservas_restaurante'
});

connection.connect((err) => {
  if (err) throw err;
  console.log('Conectado a la base de datos');
});

module.exports = connection;
```

Parte Frontend (Interfaz del usuario)

- **views – Vistas**

- Archivos *.ejs* que renderizan HTML dinámico.
- *index.ejs* es la página de reservas.
- A continuación, un pequeño fragmento del archivo donde se ve cómo configurar el campo “Personas”, el botón “Reservar” y el modal de confirmación de reserva.

```
<!-- Campo Personas -->
<div class="relative">
  <label class="block mb-1 font-semibold">Personas: <span class="text-red-500">*</span></label>
  <div class="flex items-center">
    <i class="bi bi-people-fill mr-3 text-gray-400"></i> <!-- Icono antes del placeholder -->
    <input type="number" placeholder="Número de personas" name="personas" id="personas" min="1" max="6" required
      class="border border-gray-300 rounded-lg p-2 w-full focus:outline-none focus:ring-2 focus:ring-blue-400">
  </div>
  <div class="invalid-feedback" id="errorPersonas" style="display: none; color: red;">
    Este campo es obligatorio y debe ser un número válido.
  </div>
</div>

<!-- BOTÓN "RESERVAR"-->
  <button type="submit"
    class="bg-blue-500 hover:bg-blue-600 text-white font-semibold py-2 px-4 rounded-lg w-full">
    Reservar
  </button>
</form>
</div>

<!-- Modal de Confirmación -->
<div id="modalConfirmacion" class="hidden fixed inset-0 bg-black bg-opacity-50 flex items-center justify-center z-50">
  <div class="bg-white rounded-lg shadow-lg p-8 w-80 text-center animate-fade-in">
    <h2 class="text-2xl font-bold mb-2 text-green-600">¡Reserva Confirmada!</h2>
    <p class="text-sm text-gray-500 mb-4">Datos de la reserva:</p>
    <div id="datosReserva" class="text-gray-700 mb-6 text-left whitespace-pre-line"></div>
    <button id="cerrarModal" class="bg-green-500 hover:bg-green-600 text-white font-bold py-2 px-4 rounded">
      Cerrar
    </button>
  </div>
</div>
```

- *admin.ejs* muestra información de las reservas realizadas.
- Parte del frontend porque definen lo que el usuario ve.

```

<div class="container mt-5">
  <h1 class="text-center mb-4">Lista de Reservas</h1>
  <div class="table-responsive">
    <table class="table table-bordered table-hover text-center align-middle">
      <thead class="table-dark">
        <tr>
          <th>Nombre completo</th>
          <th>ID</th>
          <th>Fecha</th>
          <th>Hora</th>
          <th>Personas</th>
        </tr>
      </thead>
      <tbody>
        <% reservas.forEach(reserva => { %>
          <tr>
            <td><%= reserva.nombre_completo %></td>
            <td><%= reserva.id %></td>
            <td>
              <%
                const partes = reserva.fecha.split('-');
                const fechaFormateada = `${partes[2]}/${partes[1]}/${partes[0]}`;
              %>
              <%= fechaFormateada %>
            </td>
            <td><%= reserva.hora %></td>
            <td><%= reserva.personas %></td>
          </tr>
        <% }} %>
      </tbody>
    </table>
  </div>
</div>

```

- **public/** – Carpeta pública para recursos estáticos

- **css/style.css** – Estilos visuales (colores, tamaños, etc.).
- **js/main.js** – Lógica de *frontend*, por ejemplo, interacción con *WebSockets* o validaciones en el cliente (*explicaremos gran parte del código de este archivo más adelante*).

Sabiendo esto, ¿cómo fluye la información?

1) Iniciamos la aplicación desde la terminal de Visual Studio Code con el comando **node app.js** (nombre del archivo principal de la app).

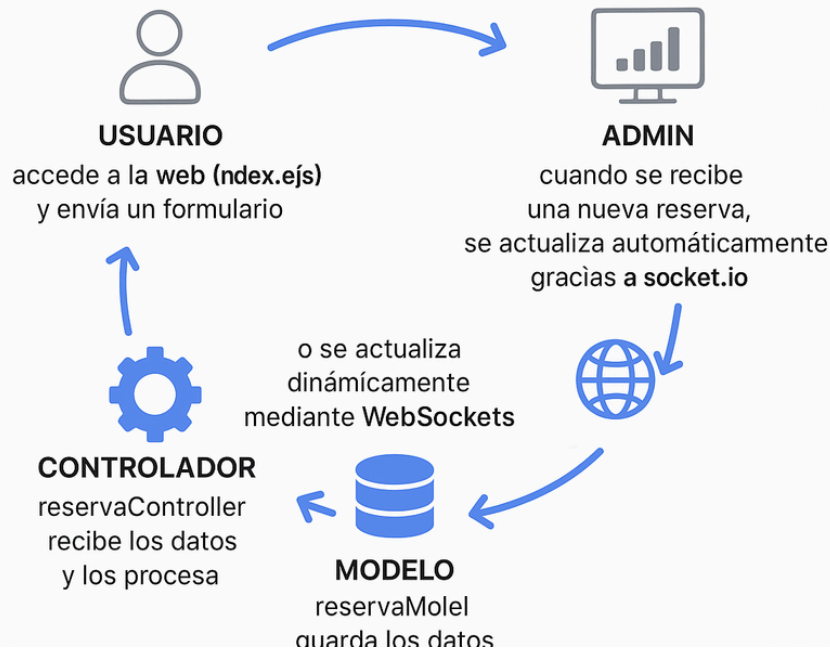
2) Abrimos la url **http://localhost:3000** en nuestro navegador. Esto es así porque estamos trabajando en local. Además la escucha es en el puerto 3000 pues así lo configuramos en el archivo principal *app.js*.

✓ TERMINAL

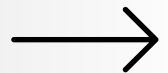
```
lidiagarciamunoz@Lidias-MacBook-Pro reserva-restaurant % node app.js  
Servidor corriendo en http://localhost:3000  
Conectado a la base de datos  
Nueva conexión WebSocket
```



Cómo fluye la información



4 - Validación y retroalimentación de formularios



Uno de los pilares clave de este formulario interactivo es la **validación de los datos ingresados por el usuario**.

Se implementan con JavaScript de manera dinámica, ofreciendo una experiencia más fluida gracias a la **validación en tiempo real** y la **retroalimentación visual inmediata**.

4.1 - Validación en tiempo real

La validación de formularios en JavaScript permite garantizar que los datos introducidos por el usuario cumplan ciertos requisitos antes de ser procesados.

En esta app, se utilizan validaciones con técnicas de interacción visual para mejorar la experiencia del usuario, tanto **automáticas (en tiempo real)** como al **enviar el formulario**.

La **validación en tiempo real** se refiere a revisar los datos del usuario **a medida que los escribe o al moverse entre campos**, sin esperar a que pulse el botón “Reservar”. Esto se implementa usando principalmente eventos como **input** y **blur**.

En nuestra app: Se usa el **evento blur** en cada campo (*input* y *select*) para validar cuando el usuario **abandona el campo** (pierde el foco):

```
// Agregamos el evento blur para validación de cada campo
nombreInput.addEventListener('blur', () => validateField(nombreInput));
fechaInputField.addEventListener('blur', () => validateField(fechaInputField));
horaSelectField.addEventListener('blur', () => validateField(horaSelectField));
personasInput.addEventListener('blur', () => validateField(personasInput));
```

Esto ayuda al usuario a corregir errores sin esperar al final, reduciendo la frustración y aumentando la tasa de formularios completados correctamente.

Creamos también la **función para validar los campos**:

```
// Función para validar los campos
function validateField(field) {
  const value = field.value.trim();
  let isValid = true;
  let errorMessage = "";
```

Si el usuario dejara el campo vacío, al pasar al siguiente campo, será advertido en tiempo real del error.

Ejemplo de validación en el campo nombre completo:


```
// Validar el campo
if (field.id === 'nombre_completo') {
  if (!value) {
    isValid = false;
    errorMessage = 'El nombre completo es obligatorio.';
  }
}
```

Funcionalidad en app :

VALIDACIÓN EN TIEMPO REAL CON CAMPOS VACÍOS


Reservar una Mesa

Nombre completo: *




El nombre completo es obligatorio.

Fecha: *




La fecha es obligatoria.

Hora: *



Selecciona una hora.

Personas: *



Por favor ingresa un número válido para las personas.

Reservar

NOTA:



Mensajes de error en tiempo real cuando el usuario va dejando los campos vacíos.

VALIDACIÓN EN TIEMPO REAL AL USUARIO SEGÚN VA RELLENANDO LOS INPUTS VACÍOS

The diagram illustrates the visual feedback process in a reservation form. It shows two states of the form, connected by a red arrow indicating a transition.

Initial State (Left):

- Nombre completo:** Lidia García
- Fecha:** Selecciona una fecha. Error message: La fecha es obligatoria.
- Hora:** Selecciona una hora. Error message: Selecciona una hora.
- Personas:** Número de personas. Error message: Por favor ingresa un número válido para las personas.
- Botón:** Reservar

Final State (Right):

- Nombre completo:** Lidia García
- Fecha:** 23/07/2025
- Hora:** 15:00
- Personas:** Número de personas. Error message: Por favor ingresa un número válido para las personas.
- Botón:** Reservar

NOTA:

Retroalimentación visual al usuario en tiempo real. Según va rellenando los campos que anteriormente dejó vacíos, van desapareciendo los mensajes de error. De esta manera, el usuario será consciente de que el error ha sido subsanado.

4.2 - Retroalimentación visual al usuario

Este apartado abarca cómo se le **comunica al usuario si los datos están bien o mal**, mediante colores, mensajes, cuadros de alerta, entre otros.

Todo esto podemos conseguirlo gracias al evento **submit**.

En nuestra aplicación, **el evento submit entra en acción cuando el usuario hace clic en el botón "Reservar" del formulario**.

El evento **submit** del formulario se dispara cuando, se validan todos los campos y el usuario presiona el botón "Registrarse":

```
formReserva.addEventListener('submit', async function (e) {  
  e.preventDefault();
```

El siguiente código comprueba la condición de campos vacíos. En caso afirmativo, salta el mensaje del *alert*.

```
// Validación final antes de enviar el formulario  
if (!nombre_completo || !fechaOriginal || !hora || !personas || isNaN(personas)) {  
  alert('Por favor, completa todos los campos correctamente.');
```

Funcionalidad en app:

VALIDACIÓN AL HACER CLIC EN EL BOTÓN “Reservar” CON CAMPOS VACÍOS

The screenshot shows a web application interface. At the top, a black alert box with a red border displays the message: "localhost:3000 dice" and "Por favor, completa todos los campos correctamente." with a yellow "Aceptar" button. Below the alert, the form contains four fields, each with a red asterisk indicating a required field:

- Nombre completo:** A text input field containing "Tu nombre completo" and a red error message "El nombre completo es obligatorio."
- Fecha:** A date picker field showing "Selecciona una fecha" and a red error message "La fecha es obligatoria."
- Hora:** A time picker field showing "Selecciona una hora" and a red error message "Selecciona una hora."
- Personas:** A text input field containing "Número de personas" and a red error message "Por favor ingresa un número válido para las personas."

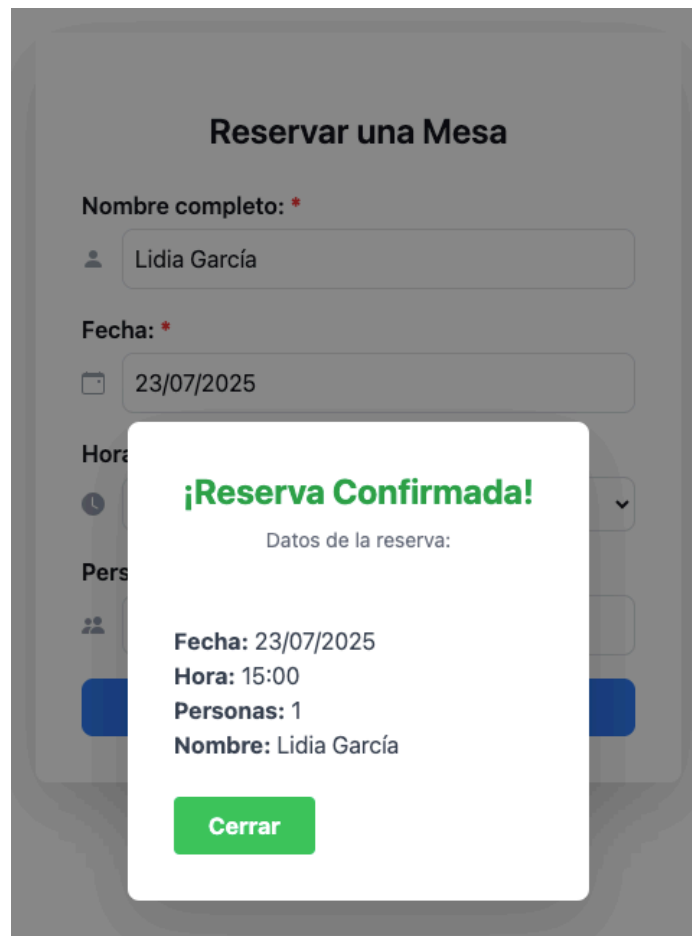
At the bottom of the form is a blue button labeled "Reservar".

NOTA:



Alerta al pulsar el botón de “Reservar” cuando uno o más campos están vacíos.

RETROALIMENTACIÓN VISUAL AL USUARIO POSITIVA



NOTA:

Retroalimentación visual al usuario cuando hace clic en el botón “Reservar” y todos los campos están rellenos correctamente, brindando así una retroalimentación clara al usuario de que su acción fue exitosa.

Esta retroalimentación positiva al usuario, se consigue gracias a un modal con estilos creados con Bootstrap.

4.3 - Vista Admin

La **vista admin** es una interfaz diseñada para que el administrador del sistema pueda **ver en tiempo real todas las reservas** que han sido enviadas desde el formulario principal por los

usuarios.

¿Cómo funciona?

- Esta vista está construida usando el motor de plantillas **EJS** (Embedded JavaScript Templates).
- Cuando el servidor recibe una solicitud a la ruta de administración, **envía un arreglo de objetos de reserva** al archivo *admin.ejs*, el cual los recorre y los muestra en una tabla.

```
<% reservas.forEach(reserva => { %>
  <tr>
    <td><%= reserva.nombre_completo %></td>
    <td><%= reserva.id %></td>
    <td>
      <%
        const partes = reserva.fecha.split('-');
        const fechaFormateada = `${partes[2]}/${partes[1]}/${partes[0]}`;
      %>
      <%= fechaFormateada %>
    </td>
    <td><%= reserva.hora %></td>
    <td><%= reserva.personas %></td>
  </tr>
<% }) %>
```

Este código:

- 1) **Itera sobre el arreglo reservas** que proviene del servidor.
- 2) Por cada reserva muestra el **nombre del cliente**, el **ID**, la **fecha** (reformateada a dd/mm/yyyy), la **hora** y la **cantidad de personas**.
- 3) Todo esto se renderiza dentro de una **tabla con estilo de Bootstrap**, lo que facilita su visualización y organización para el administrador.

Funcionalidad en app :

ACTUALIZACIÓN TABLA INFORMATIVA DE RESERVAS

Lista de Reservas

Nombre completo	ID	Fecha	Hora	Personas
Fran Costa	26	11/06/2025	15:00:00	4
Sergio Sánchez	28	25/07/2025	15:00:00	2

¡Reserva Confirmada!

Datos de la reserva:

Fecha: 23/07/2025
Hora: 15:00
Personas: 1
Nombre: Lidia García

Cerrar

Lista de Reservas

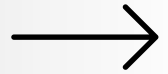
Nombre completo	ID	Fecha	Hora	Personas
Fran Costa	26	11/06/2025	15:00:00	4
Lidia García	29	23/07/2025	15:00:00	1
Sergio Sánchez	28	25/07/2025	15:00:00	2

NOTA:

Permite **gestionar y controlar reservas** de manera visual y centralizada. Es ideal para que el negocio pueda **ver rápidamente qué mesas están reservadas**, cuántas personas asistirán y cuándo.



5 - Modificación dinámica de formularios



Una de las principales fortalezas de esta aplicación es su capacidad de interactuar con el servidor sin necesidad de recargar la página, ofreciendo una experiencia fluida y moderna al usuario. Esto se logra mediante dos tecnologías clave: **AJAX/Fetch** y **WebSockets**.

5.1 AJAX + Fetch

AJAX (Asynchronous JavaScript and XML) es una técnica que permite a las aplicaciones web comunicarse con el servidor de manera **asíncrona**, es decir, sin recargar la página por completo. Esto mejora la velocidad de la aplicación y la experiencia del usuario.

Por otro lado, **fetch** es una API moderna de JavaScript que implementa AJAX de forma más sencilla y limpia. Permite enviar o recibir datos del servidor usando peticiones HTTP como GET, POST, etc.

¿Cómo lo implementamos en nuestra aplicación?

En nuestra aplicación, se utiliza *fetch* para enviar los datos del formulario de reserva al servidor sin que el usuario tenga que abandonar o recargar la página.

```
const respuesta = await fetch('/reservar', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: datos
});
```

5.2 WebSockets

WebSockets son un protocolo que permite una **comunicación bidireccional** y en **tiempo real** entre el cliente (navegador) y el servidor. A diferencia de las peticiones HTTP tradicionales (como *Fetch*), los *WebSockets* abren una conexión persistente que se mantiene activa.

¿Qué es Socket.IO?

Socket.IO es una **librería que facilita el uso de WebSockets**, ofreciendo una forma simple de enviar y recibir eventos entre cliente y servidor.

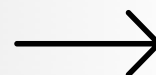
¿Dónde se usa en la app?

Utilizamos Socket.IO para mantener la **vista admin** actualizada en tiempo real. Cada vez que se hace una nueva reserva, el *backend* emite un evento a todos los clientes conectados (en el caso de la app, a la tabla informativa de reservas de la vista *admin*).

En el backend

```
// WebSockets
io.on('connection', (socket) => {
  console.log('Nueva conexión WebSocket');
  socket.on('nuevaReserva', () => {
    io.emit('actualizarReservas');
  });
});
```

6 - Conclusiones



Este trabajo demuestra la **importancia del uso de técnicas de validación robustas para mejorar la experiencia del usuario** y asegurar la integridad de los datos.

Como puntos a destacar de esta sencilla app de reserva para restaurante destacamos los siguientes puntos:

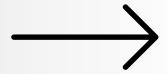


- **Experiencia de usuario optimizada:** La interfaz es intuitiva y amigable, con validaciones en tiempo real que mejoran la interacción y reducen errores, permitiendo a los usuarios completar una reserva de forma rápida y confiable.
- **Comunicación en tiempo real:** Gracias a la integración de WebSockets y bases de datos en tiempo real. Tanto el cliente como el administrador reciben actualizaciones instantáneas, lo cual es esencial en un entorno como el de la restauración, donde la disponibilidad cambia constantemente.
- **Arquitectura organizada y mantenible:** La adopción del patrón MVC (Modelo-Vista-Controlador) ha permitido estructurar el proyecto de manera clara y coherente, lo que facilita futuras ampliaciones o modificaciones del sistema.
- **Gestión eficiente:** La vista *admin* permite al restaurante tener control total sobre sus reservas y actualizaciones de contenido, lo cual mejora significativamente la eficiencia operativa.
- **Base para futuras mejoras:** Esta aplicación sienta las bases para futuras funcionalidades como estadísticas de reservas, sistema de fidelización o notificaciones *push*, lo que demuestra su potencial como solución integral para la gestión de restaurantes.



En resumen, este proyecto no solo refleja la parte técnica de validar formularios, sino también el esfuerzo por ofrecer una experiencia fluida, accesible y profesional para el usuario final.

7 - Fuentes externas y recursos utilizados



- **Cesur, Desarrollo web entorno servidor** - “Generación dinámica de páginas web interactivas”.
- **Lidia García Muñoz** – “Formulario web interactivo: validaciones en tiempo real, expresiones regulares y retroalimentación visual.”
- **Lidia García Muñoz** – “Desarrollando una aplicación web interactiva.”

