GUÍA TÉCNICA DE HOGARFIX - PARTE 1

Manual Completo para Entender y Explicar tu Proyecto

¿QUÉ ES HOGARFIX?

Imagina que se te rompió una tubería en casa a las 10 de la noche. ¿A quién llamas? ¿Cómo sabes si es confiable? ¿Cuánto te va a cobrar? **HogarFix resuelve exactamente ese problema.**

HogarFix es una **plataforma web** que conecta a personas que necesitan servicios para el hogar (clientes) con profesionales que ofrecen esos servicios (plomeros, electricistas, carpinteros, etc.). Es como un "marketplace" o mercado digital donde se encuentran la oferta y la demanda de servicios domésticos.

Analogías Simples para Entender HogarFix:

Como Uber para el hogar:

- Uber conecta pasajeros con conductores → HogarFix conecta clientes con profesionales
- En Uber ves la calificación del conductor → En HogarFix ves las estrellas del profesional
- En Uber pagas desde la app → En HogarFix el profesional recibe su pago automáticamente

Como un Centro Comercial Digital:

- Tienes muchas tiendas (profesionales) en un solo lugar
- Puedes comparar precios y calificaciones antes de elegir
- Hay un administrador (admin) que cuida que todo funcione bien

Como una Red Social, pero de Servicios:

- Te registras con tu email
- Tienes un perfil personalizado
- Puedes chatear con otros usuarios
- Dejas reseñas y calificaciones

ARQUITECTURA DEL PROYECTO

¿Qué significa "arquitectura"?

La arquitectura es cómo está organizado tu proyecto. Es como cuando construyes una casa: necesitas planos que muestren dónde va cada habitación, cómo se conectan, y qué función cumple cada espacio.

En programación, la "arquitectura" responde a estas preguntas:

- ¿Qué partes tiene mi aplicación?
- ¿Cómo se comunican entre sí?
- ¿Dónde está cada cosa?
- ¿Quién hace qué?

Las Tres Capas de HogarFix (Como un Restaurante):

Piensa en HogarFix como un restaurante de tres niveles:

```
┌─────────────────────────────────────┐
│         FRONTEND (Cliente)          │  ← EL COMEDOR
│   HTML + CSS + JavaScript           │    Lo que el cliente VE
│   (Navegador web)                   │    (La carta, las mesas, la decoración)
└─────────────────────────────────────┘
              ↕  (HTTP/HTTPS - El mesero lleva pedidos)
┌─────────────────────────────────────┐
│         BACKEND (Servidor)          │  ← LA COCINA
│   Node.js + Express                 │    Donde se PREPARA todo
│   (Lógica de negocio)               │    (Chefs que cocinan)
└─────────────────────────────────────┘
              ↕  (SQL - Busca ingredientes)
┌─────────────────────────────────────┐
│    BASE DE DATOS (Almacenamiento)   │  ← LA DESPENSA
│          SQLite                     │    Donde se GUARDA todo
│   (Datos persistentes)              │    (Ingredientes organizados)
└─────────────────────────────────────┘
```

Explicación más detallada:

1. **FRONTEND (La Cara Bonita):**
    - Es TODO lo que ves en tu navegador
    - Los botones, colores, formularios, imágenes
    - Es como la interfaz de Instagram o Facebook
    - Tecnologías: HTML (estructura), CSS (diseño), JavaScript (interactividad)
2. **BACKEND (El Cerebro):**
    - Es lo que NO ves pero que hace el trabajo pesado
    - Verifica contraseñas, guarda pedidos, calcula precios
    - Es como el motor de un auto: no lo ves, pero sin él nada funciona
    - Tecnología: Node.js con Express
3. **BASE DE DATOS (La Memoria):**
    - Guarda TODA la información de forma permanente
    - Usuarios, pedidos, mensajes, calificaciones
    - Es como el disco duro de tu computadora
    - Tecnología: SQLite (un archivo llamado database.db)

🗀 ESTRUCTURA DE CARPETAS

¿Por qué está organizado así?

```
TrabajodeSoftware/
│
├── frontend/              ← Todo lo que ve el usuario
│   ├── index.html         ← Página principal
│   ├── login.html         ← Iniciar sesión
│   ├── register.html      ← Registrarse
│   ├── panel-cliente.html      ← Panel del cliente
│   ├── panel-profesional.html  ← Panel del profesional
│   ├── panel-admin.html        ← Panel del administrador
│   ├── profesionales.html      ← Buscar profesionales
│   ├── profile.html           ← Ver perfil
│   ├── CSS/
│   │   └── style.css      ← Estilos visuales
│   └── js/
│       ├── auth.js        ← Manejo de login/registro
│       ├── panelCliente.js    ← Lógica panel cliente
│       ├── panelProfesional.js ← Lógica panel profesional
│       └── ...
│
└── backend/               ← Todo lo que procesa en el servidor
    ├── server.js          ← Archivo principal del servidor
    ├── package.json       ← Dependencias del proyecto
    └── src/
        ├── config/
        │   └── database.js    ← Configuración BD
        ├── controllers/       ← Controlan las peticiones
        ├── middleware/        ← Filtros de seguridad
        ├── models/           ← Estructura de datos
        ├── repositories/     ← Acceso a la base de datos
        └── services/         ← Lógica de negocio
```

🎨 FRONTEND - La Cara Visible

¿Qué es el Frontend?

Es todo lo que el usuario **ve y con lo que interactúa** en su navegador: botones, formularios, colores, textos, imágenes.

Tecnologías Utilizadas:

1. HTML5 (HyperText Markup Language)

¿Qué es en palabras simples? HTML es el "esqueleto" de una página web. Define QUÉ elementos existen, pero NO cómo se ven.

Analogía de la Casa: Si tu página web fuera una casa, HTML serían los ladrillos, las paredes, las ventanas, las puertas. Define la estructura, pero sin pintura ni decoración.

Ejemplo del Mundo Real: Cuando ves un formulario de login en HogarFix, el HTML dice:

- "Aquí va un título que dice 'Iniciar Sesión'"
- "Aquí va una caja para escribir el email"
- "Aquí va una caja para escribir la contraseña"
- "Aquí va un botón que dice 'Entrar'"

Código Real de HogarFix:

```
<!-- Esto crea el formulario de login -->
<h1>Bienvenido a HogarFix</h1>
<input type="text" placeholder="Tu email">
<input type="password" placeholder="Tu contraseña">
<button>Iniciar Sesión</button>
```

Pregunta típica: "¿Y si quiero que el botón sea azul?" **Respuesta:** HTML NO controla colores. Para eso existe CSS.

---

2. CSS3 (Cascading Style Sheets)

¿Qué es en palabras simples? CSS es el "maquillaje" de la página web. Define CÓMO se ve todo: colores, tamaños, posiciones, animaciones.

Analogía de la Casa: Siguiendo con la casa, CSS es la pintura de las paredes, el tipo de piso, los muebles, las cortinas. Todo lo que hace que la casa se vea bonita.

Ejemplo del Mundo Real: El mismo botón del ejemplo anterior, ahora con estilo:

Código Real de HogarFix:

```
/* Esto hace que el botón se vea profesional */
.boton-principal {
    background-color: #007bff;  /* Fondo azul */
    color: white;               /* Letras blancas */
    padding: 10px 20px;         /* Espacio interno (ancho y alto) */
    border-radius: 5px;         /* Esquinas redondeadas */
    font-size: 16px;            /* Tamaño de letra */
    cursor: pointer;            /* Manita al pasar el mouse */
}

/* Cuando pasas el mouse encima, se oscurece */
.boton-principal:hover {
    background-color: #0056b3;  /* Azul más oscuro */
}
```

Pregunta típica: "¿Por qué hay un punto antes de 'boton-principal'?" **Respuesta:** El punto indica que es una "clase". En HTML escribes `class="boton-principal"` y CSS lo busca con `.boton-principal`

---

3. JavaScript (ES6+)

¿Qué es en palabras simples? JavaScript es el "sistema nervioso" de la página. Hace que TODO sea INTERACTIVO. Sin JavaScript, tu página sería como una revista: solo puedes ver, pero no interactuar.

Analogía de la Casa: JavaScript es todo el sistema eléctrico: cuando presionas un interruptor, se enciende la luz. Cuando abres la llave, sale agua. Es lo que hace que las cosas FUNCIONEN.

Ejemplo del Mundo Real - Paso a Paso:

Situación: Un usuario hace clic en "Solicitar Servicio"

1. **HTML** muestra el botón
2. **CSS** lo hace verse bonito
3. **JavaScript** hace que PASE ALGO cuando le das clic

Código Real de HogarFix:

```
// 1. Espera a que la página cargue completamente
document.addEventListener('DOMContentLoaded', function() {

    // 2. Busca el botón que tiene id="btnSolicitar"
    const boton = document.getElementById('btnSolicitar');

    // 3. Cuando alguien haga clic en ese botón...
    boton.addEventListener('click', function() {

        // 4. Obtiene lo que el usuario escribió
        const descripcion = document.getElementById('descripcion').value;

        // 5. Verifica que no esté vacío
        if (descripcion === '') {
            alert('¡Debes escribir una descripción!');
            return;  // Se detiene aquí
        }

        // 6. Si todo está bien, envía al servidor
        enviarSolicitud(descripcion);

        // 7. Muestra mensaje de confirmación
        alert('¡Solicitud enviada exitosamente!');
    });
});
```

Desglose para entender cada línea:

```
document.addEventListener('DOMContentLoaded', function() { ... });
```

↑ "Espera a que TODO el HTML esté cargado antes de ejecutar este código" ¿Por qué? Porque si intentas buscar un botón que aún no existe, falla.

```
const boton = document.getElementById('btnSolicitar');
```

↑ "Busca en todo el HTML un elemento que tenga id='btnSolicitar' y guárdalo en la variable 'boton'" Es como buscar un libro específico en una biblioteca.

```
boton.addEventListener('click', function() { ... });
```

↑ "Cuando alguien haga CLIC en ese botón, ejecuta esta función" Es como programar una alarma: "Cuando sean las 7am, suena"

```
const descripcion = document.getElementById('descripcion').value;
```

↑ "Busca el input con id='descripcion' y obtén lo que el usuario escribió" `.value` es la propiedad que contiene el texto.

Pregunta típica: "¿Por qué a veces dice 'function' y a veces '=>'?" **Respuesta:** Son dos formas de escribir funciones:

```
// Forma tradicional
function saludar() {
    return "Hola";
}

// Forma moderna (arrow function)
const saludar = () => {
    return "Hola";
}
```

Hacen lo mismo, la segunda es más corta.

---

4. Bootstrap 5.3.2

¿Qué es en palabras simples? Bootstrap es una "biblioteca de componentes pre-diseñados". Es como tener una caja de LEGO: en lugar de tallar cada pieza desde cero, usas piezas ya hechas.

Analogía de la Ropa: Hacer una página sin Bootstrap = Coser tu propia ropa desde cero Usar Bootstrap = Comprar ropa en una tienda

Ambos te dan ropa, pero el segundo es MUCHO más rápido.

Ejemplo Práctico - Botones:

SIN Bootstrap (tienes que escribir TODO el CSS):

```
<button class="mi-boton">Aceptar</button>

<style>
.mi-boton {
    background-color: #28a745;
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 4px;
    font-size: 14px;
    cursor: pointer;
    transition: background-color 0.3s;
}
.mi-boton:hover {
    background-color: #218838;
}
</style>
```

CON Bootstrap (solo una clase):

```
<button class="btn btn-success">Aceptar</button>
```

↑ ¡Y ya está! Bootstrap tiene TODO el CSS pre-programado.

Componentes que Bootstrap te regala:

- Botones de colores: `btn-primary` (azul), `btn-success` (verde), `btn-danger` (rojo)
- Formularios bonitos con validación
- Tarjetas (cards) para mostrar información
- Menús de navegación responsive
- Modales (ventanas emergentes)
- Alerts (mensajes de advertencia/éxito)
- Grids (sistema de columnas para organizar contenido)

Ejemplo en HogarFix - Sistema de Columnas:

```
<!-- Divide la pantalla en 2 columnas iguales -->
<div class="row">
    <div class="col-6">
        Columna izquierda (50% del ancho)
    </div>
    <div class="col-6">
        Columna derecha (50% del ancho)
```

```
    </div>
</div>
```

Pregunta típica: "¿Por qué usamos Bootstrap y no diseñamos todo nosotros?" **Respuesta:** Porque Bootstrap:

1. Ahorra MUCHO tiempo
2. Está probado por millones de desarrolladores (menos bugs)
3. Es "responsive" automáticamente (funciona en móviles)
4. Se ve profesional sin ser diseñador gráfico

🎯 RESUMEN DE FRONTEND:

Tecnología	Función	Analogía
HTML	Estructura (QUÉ hay)	Esqueleto de la casa
CSS	Diseño (CÓMO se ve)	Pintura y decoración
JavaScript	Interactividad (QUÉ hace)	Sistema eléctrico
Bootstrap	Componentes listos	LEGO pre-armado

Todos trabajan juntos:

```
<!-- HTML: Define un botón -->
<button id="miBoton" class="btn btn-primary">
    Haz clic aquí
</button>

<!-- CSS: Lo personaliza más -->
<style>
#miBoton {
    font-weight: bold;
}
</style>

<!-- JavaScript: Lo hace funcionar -->
<script>
document.getElementById('miBoton').addEventListener('click', () => {
    alert('¡Funcionó!');
});
</script>
```

🔧 BACKEND - El Motor Invisible

¿Qué es el Backend?

Imagina que HogarFix es un iceberg:

- **Frontend** = La parte que ves flotando (10% del iceberg)
- **Backend** = La parte sumergida gigante que no ves (90% del iceberg)

El backend es TODO lo que sucede "detrás de bambalinas":

- Verificar si tu contraseña es correcta
- Guardar un pedido en la base de datos
- Enviar notificaciones
- Calcular el saldo de un profesional
- Crear respaldos automáticos

Comparación con un Cajero Automático:

- Tú ves la pantalla y los botones (frontend)
- Pero NO ves cómo verifica tu PIN, consulta tu saldo, o cuenta los billetes (backend)

Tecnologías Utilizadas:

---

1. Node.js - El Motor de JavaScript

¿Qué es en palabras simples? Node.js permite que JavaScript (que normalmente solo funciona en navegadores) también funcione en SERVIDORES.

Analogía del Idioma:

- JavaScript en navegador = Hablar español solo en México
- Node.js = Hablar español en TODO el mundo

Antes de Node.js, tenías que aprender DOS lenguajes:

- JavaScript para frontend (lo que ve el usuario)
- PHP/Java/Python para backend (el servidor)

Con Node.js usas JavaScript para AMBOS. Es como ser bilingüe en un solo idioma.

¿Por qué es útil?

1. **Un solo lenguaje:** Frontend y backend hablan el mismo idioma
2. **Rápido:** Node.js puede manejar muchas peticiones al mismo tiempo (asíncrono)
3. **Popular:** Millones de bibliotecas disponibles en NPM
4. **Comunidad:** Si tienes un problema, alguien ya lo resolvió en StackOverflow

Analogía del Restaurante:

- Sin Node.js = El mesero habla español, el chef habla japonés (complicado)
- Con Node.js = Todos hablan español (comunicación fluida)

Ejemplo Real:

```
// Este código funciona tanto en navegador como en servidor gracias a Node.js
const sumar = (a, b) => a + b;
```

```
console.log(sumar(5, 3));  // 8
```

---

2. Express.js - El Framework Mágico

¿Qué es en palabras simples? Express.js es un "atajo" para crear servidores web. Sin Express, crear un servidor tomaría 100 líneas de código. Con Express, solo 10.

Analogía del Restaurante (otra vez): Express.js es como el **maitre** (jefe de meseros) del restaurante:

```
Cliente dice: "Quiero ver el menú"
  ↓
Maitre (Express): "Ok, te llevo a la mesa 5"
  ↓
Mesero específico atiende esa mesa
  ↓
Cliente dice: "Quiero ordenar pizza"
  ↓
Maitre: "Esa orden va a la cocina italiana"
  ↓
Chef italiano hace la pizza
```

En código, se ve así:

```
const express = require('express');  // Importar Express
const app = express();  // Crear el "maitre"

// RUTA 1: Cuando alguien visita /api/menu
app.get('/api/menu', (req, res) => {
    res.json({ platillos: ['Pizza', 'Pasta', 'Ensalada'] });
});

// RUTA 2: Cuando alguien visita /api/pedidos
app.get('/api/pedidos', (req, res) => {
    res.json({ pedidos: ['Pedido #1', 'Pedido #2'] });
});

// RUTA 3: Cuando alguien ENVÍA un nuevo pedido
app.post('/api/pedidos', (req, res) => {
    const nuevoPedido = req.body;  // Lo que el cliente envió
    // ... guardar en base de datos
    res.json({ mensaje: '¡Pedido recibido!' });
});

// Iniciar el servidor en el puerto 3000
app.listen(3000, () => {
    console.log('Restaurante abierto en http://localhost:3000');
});
```

Desglose línea por línea:

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

↑ "Ve a la carpeta node_modules, busca la biblioteca 'express', y tráela aquí" Es como decir "Necesito el control remoto de la TV"

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

↑ "Crea un nuevo servidor web y guárdalo en la variable 'app'" Ahora 'app' es tu servidor, listo para recibir visitantes

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

↑ Desglose completo:

- `app.get` = Cuando alguien haga una petición GET (leer datos)
- `/api/menu` = A esta dirección específica
- `(req, res)` = Dos objetos mágicos:
  - `req` (request) = Lo que el cliente PIDIÓ
  - `res` (response) = Lo que tú vas a RESPONDER
- `=> { ... }` = Ejecuta este código

```
res.json({ platillos: ['Pizza', 'Pasta'] });
```

↑ "Responde enviando este objeto en formato JSON" JSON es como el "idioma universal" para enviar datos

Pregunta típica: "¿Qué es req y res?" **Respuesta:**

- `req` (request) = El sobre que te llegó por correo (contiene la petición)
- `res` (response) = El sobre que tú envías de vuelta (contiene la respuesta)

Ejemplo Real en HogarFix:

```
// En server.js
app.get('/api/profesionales', authMiddleware, async (req, res) => {
    try {
        // 1. Obtener profesionales de la base de datos
        const profesionales = await userRepository.findByRol('profesional');

        // 2. Enviar como respuesta
        res.json(profesionales);
```

```
    } catch (error) {
        // 3. Si algo salió mal, enviar error
        res.status(500).json({ error: 'Error al obtener profesionales' });
    }
});
```

¿Qué hace este código paso a paso?

1. Alguien visita `http://localhost:3000/api/profesionales`
2. Primero pasa por `authMiddleware` (verifica que tenga permiso)
3. Si tiene permiso, ejecuta la función
4. Va a la base de datos y busca todos los profesionales
5. Los envía como JSON al navegador
6. Si algo falla, envía un mensaje de error

---

3. SQLite - La Memoria del Sistema

¿Qué es en palabras simples? SQLite es una **base de datos** que guarda toda la información en un solo archivo llamado `database.db`. Es como un Excel gigante, pero mucho más potente.

Analogía del Archivero: Imagina un archivero gigante con muchos cajones:

- Cajón 1 = Usuarios (tarjetas con nombre, email, contraseña)
- Cajón 2 = Pedidos (tarjetas con descripción, precio, estado)
- Cajón 3 = Mensajes (tarjetas con texto de los chats)
- Cajón 4 = Calificaciones (tarjetas con estrellas y comentarios)

Cada cajón es una **TABLA**. Cada tarjeta es un **REGISTRO** (también llamado "fila" o "row"). Cada dato de la tarjeta es una **COLUMNA** (nombre, email, etc.).

¿Qué guarda HogarFix en la base de datos?

Tabla	¿Qué guarda?	Ejemplo
`usuarios`	Personas registradas	Juan Pérez, cliente
`pedidos`	Solicitudes de servicio	"Reparar fuga", $500
`mensajes`	Chat entre usuario y profesional	"¿A qué hora vienes?"
`categorias`	Tipos de servicios	Plomería, Electricidad
`ratings`	Calificaciones	5 estrellas, "Excelente"

Ejemplo de Consulta SQL (el lenguaje de las bases de datos):

```
-- Ver todos los usuarios
SELECT * FROM usuarios;

-- Buscar un usuario por email
```

```
SELECT * FROM usuarios WHERE email = 'cliente@test.com';

-- Crear un nuevo usuario
INSERT INTO usuarios (nombre, email, password, rol)
VALUES ('Juan', 'juan@test.com', 'hash123', 'cliente');

-- Actualizar el saldo de un profesional
UPDATE usuarios SET saldo = saldo + 500 WHERE id = 5;

-- Eliminar un pedido
DELETE FROM pedidos WHERE id = 10;
```

Desglose de una consulta:

```
SELECT * FROM usuarios WHERE email = 'juan@test.com';
```

- `SELECT` = "Quiero ver"
- `*` = "Todo" (todas las columnas)
- `FROM usuarios` = "De la tabla de usuarios"
- `WHERE email = 'juan@test.com'` = "Solo donde el email sea este"

Es como decir: "Muéstrame todo del archivero de usuarios, pero solo la tarjeta que dice juan@test.com"

¿Por qué SQLite y no MySQL/PostgreSQL?

Característica	SQLite	MySQL/PostgreSQL
Complejidad	Muy simple	Más complejo
Archivo	Un solo archivo .db	Servidor separado
Velocidad	Rápido para apps pequeñas	Más rápido para millones de datos
Instalación	No requiere instalación	Requiere instalar servidor
Ideal para	Prototipos, apps pequeñas	Aplicaciones grandes, producción

Para HogarFix, SQLite es perfecto porque:

1. Es una demostración/prototipo
2. Facilita el despliegue (solo un archivo)
3. Es rápido para cientos/miles de usuarios
4. Si crece, migrar a PostgreSQL es fácil (gracias al patrón Repository)

Cómo se ve el archivo database.db:

```
backend/
  ├── database.db  ← Todo está aquí (1 MB - 10 MB)
  ├── server.js
  └── ...
```

Si abres `database.db` con un editor (DB Browser for SQLite), verías algo así:

```
Tabla: usuarios
┌────┬──────────┬──────────────────┬──────────┬──────────────┐
│ id │  nombre  │      email       │   rol    │    saldo     │
├────┼──────────┼──────────────────┼──────────┼──────────────┤
│ 1  │ Admin    │ admin@hogar...   │ admin    │ 0.00         │
│ 2  │ Juan     │ juan@test.com    │ prof...  │ 1500.00      │
│ 3  │ María    │ maria@test.com   │ cliente  │ 0.00         │
└────┴──────────┴──────────────────┴──────────┴──────────────┘
```

Pregunta típica: "¿Qué pasa si se borra database.db?" **Respuesta:** Se pierde TODO. Por eso HogarFix hace respaldos automáticos cada 24 horas en la carpeta `/backups`.

RESUMEN DE BACKEND:

Tecnología	Función	Analogía
Node.js	Ejecutar JavaScript en servidor	Motor del auto
Express.js	Manejar rutas y peticiones	Maitre del restaurante
SQLite	Guardar datos permanentemente	Archivero gigante

Flujo Completo Backend:

```
1. Usuario visita /api/pedidos
   ↓
2. Express recibe la petición
   ↓
3. Verifica autenticación (middleware)
   ↓
4. Controller procesa la petición
   ↓
5. Repository consulta SQLite
   ↓
6. SQLite devuelve los datos
   ↓
7. Express envía respuesta al usuario
```

AUTENTICACIÓN Y SEGURIDAD

¿Qué es la Autenticación?

Es el proceso de verificar que un usuario es quien dice ser (login con email y contraseña).

JWT (JSON Web Tokens)

¿Qué es? Un "carnet de identidad digital" que el servidor le da al usuario después de iniciar sesión.

¿Cómo funciona en HogarFix?

1. **Usuario se registra o inicia sesión:**

```
POST /api/auth/login
Body: { email: "cliente@test.com", password: "123456" }
```

2. **Servidor verifica credenciales y genera un TOKEN:**

```
const token = jwt.sign(
    { id: usuario.id, rol: usuario.rol },  // Datos del usuario
    'clave_secreta',                       // Llave secreta
    { expiresIn: '24h' }                   // Expira en 24 horas
);
```

3. **Cliente guarda el token y lo envía en cada petición:**

```
sessionStorage.setItem('token', token);

// En las siguientes peticiones:
fetch('/api/pedidos', {
    headers: {
        'Authorization': `Bearer ${token}`
    }
});
```

4. **Servidor verifica el token antes de responder:**

```
function authMiddleware(req, res, next) {
    const token = req.headers.authorization?.split(' ')[1];
    if (!token) return res.status(401).json({ error: 'No autorizado' });

    jwt.verify(token, 'clave_secreta', (err, decoded) => {
        if (err) return res.status(403).json({ error: 'Token inválido' });
        req.user = decoded;  // Guarda datos del usuario en la petición
        next();  // Continúa con la siguiente función
    });
}
```

Analogía: El token es como tu credencial de estudiante. La obtienes al inscribirte (login), y debes mostrarla cada vez que entras a la universidad (hacer peticiones al servidor).

🎭 SISTEMA DE ROLES

HogarFix tiene 3 tipos de usuarios, cada uno con permisos diferentes:

1. **Cliente**

¿Qué puede hacer?

- Buscar profesionales por categoría
- Solicitar servicios
- Chatear con profesionales
- Calificar servicios completados
- Ver historial de pedidos

Restricciones:

- NO puede aprobar profesionales
- NO puede ver estadísticas globales

2. **Profesional**

¿Qué puede hacer?

- Ver pedidos que le asignaron
- Confirmar precio de servicios
- Marcar pedidos como completados
- Chatear con clientes
- Ver su saldo acumulado

Restricciones:

- NO puede solicitar servicios a otros
- NO puede gestionar usuarios

3. **Administrador**

¿Qué puede hacer?

- Aprobar o rechazar profesionales
- Ver estadísticas completas (usuarios, pedidos, ingresos)
- Crear respaldos de la base de datos
- Gestionar reclamaciones
- Ver todos los usuarios y pedidos

Credenciales por defecto:

```
Email: admin@hogarfix.local
Password: admin123
```

¿Cómo se implementa?

```
// Middleware que solo permite administradores
function adminMiddleware(req, res, next) {
    if (req.user.rol !== 'admin') {
        return res.status(403).json({ error: 'Solo administradores' });
    }
    next();
}

// Ruta protegida
app.get('/api/admin/estadisticas', authMiddleware, adminMiddleware, (req, res) =>
{
    // Solo llega aquí si está autenticado Y es admin
    const stats = obtenerEstadisticas();
    res.json(stats);
});
```

FLUJO DE DATOS

Ejemplo: Un Cliente Solicita un Servicio

```
1. FRONTEND (panel-cliente.html)
   Usuario llena formulario y hace clic en "Solicitar"
   ↓

2. JAVASCRIPT (panelCliente.js)
   Captura los datos del formulario:
   {
     profesionalId: 5,
     descripcion: "Reparar fuga en baño",
     fecha: "2025-11-25"
   }
   ↓

3. PETICIÓN HTTP (fetch)
   POST /api/pedidos
   Headers: { Authorization: "Bearer eyJhbGc..." }
   Body: { profesionalId: 5, descripcion: "..." }
   ↓

4. BACKEND (server.js)
   Recibe la petición en la ruta /api/pedidos
   ↓

5. MIDDLEWARE (authMiddleware)
   Verifica que el token sea válido
   ↓
```

```
6. CONTROLLER (pedidoController.js)
   Valida los datos (¿profesional existe? ¿descripción válida?)
   ↓

7. SERVICE (pedidoService.js)
   Aplica lógica de negocio (estado inicial: "pendiente")
   ↓

8. REPOSITORY (pedidoRepository.js)
   Ejecuta consulta SQL:
   INSERT INTO pedidos (cliente_id, profesional_id, descripcion, estado)
   VALUES (3, 5, 'Reparar fuga...', 'pendiente')
   ↓

9. BASE DE DATOS (database.db)
   Guarda el pedido con ID #42
   ↓

10. RESPUESTA HTTP
    Backend responde:
    { success: true, pedidoId: 42 }
    ↓

11. FRONTEND ACTUALIZA
    Muestra mensaje: "¡Pedido enviado exitosamente!"
    Recarga la lista de pedidos
```

🧩 PATRONES DE DISEÑO UTILIZADOS

1. **MVC (Modelo-Vista-Controlador)** - Modificado

¿Qué es? Una forma de organizar el código separando responsabilidades.

```
MODELO (models/)
  ↓ Define la estructura de los datos
  Usuario: { id, nombre, email, rol, ... }

CONTROLADOR (controllers/)
  ↓ Recibe peticiones y coordina respuestas
  userController.js: registrar(), login(), obtenerPerfil()

VISTA (frontend/)
  ↓ Muestra los datos al usuario
  login.html, panel-cliente.html
```

Beneficio: Si necesitas cambiar cómo se ve el login, solo editas el HTML. Si cambias la lógica de autenticación, solo editas el controlador.

2. **Repository Pattern**

¿Qué es? Separar la lógica de acceso a datos en archivos especializados.

Sin Repository:

```
// En el controller (MAL ❌)
app.get('/api/usuarios', (req, res) => {
    db.all('SELECT * FROM usuarios', (err, rows) => {
        res.json(rows);
    });
});
```

Con Repository:

```
// userRepository.js
class UserRepository {
    getAllUsers() {
        return new Promise((resolve, reject) => {
            db.all('SELECT * FROM usuarios', (err, rows) => {
                if (err) reject(err);
                resolve(rows);
            });
        });
    }
}

// userController.js (BIEN ☑)
async function obtenerUsuarios(req, res) {
    const usuarios = await userRepository.getAllUsers();
    res.json(usuarios);
}
```

Beneficio: Si cambias de SQLite a MySQL, solo modificas el repository, no todos los controllers.

3. **Middleware Pattern**

¿Qué es? Funciones que se ejecutan ANTES de llegar al destino final.

```
// Flujo de una petición:
Cliente → authMiddleware → adminMiddleware → Controller → Respuesta
           ↓                ↓                  ↓
        ¿Token OK?      ¿Es admin?        Ejecuta acción
```

Ejemplo:

```
app.get('/api/admin/usuarios',
    authMiddleware,      // 1. Verifica token
```

```
    adminMiddleware,     // 2. Verifica rol admin
    userController.getAll // 3. Obtiene usuarios
);
```

COMUNICACIÓN FRONTEND-BACKEND

¿Cómo hablan entre sí?

Usan **HTTP/HTTPS** con el formato **REST API**.

REST API - ¿Qué significa?

REST = Representational State Transfer

Es una convención para crear URLs que representen acciones:

Método HTTP	URL	Acción
GET	/api/pedidos	Obtener todos los pedidos
GET	/api/pedidos/5	Obtener pedido #5
POST	/api/pedidos	Crear nuevo pedido
PUT	/api/pedidos/5	Actualizar pedido #5
DELETE	/api/pedidos/5	Eliminar pedido #5

Ejemplo de Petición en HogarFix:

```
// Frontend (JavaScript)
async function obtenerPedidos() {
    const response = await fetch('http://localhost:3000/api/pedidos', {
        method: 'GET',
        headers: {
            'Authorization': `Bearer ${sessionStorage.getItem('token')}`
        }
    });

    const pedidos = await response.json();
    console.log(pedidos);  // Array de pedidos
}
```

```
// Backend (Express)
app.get('/api/pedidos', authMiddleware, async (req, res) => {
    try {
        const pedidos = await pedidoRepository.getPedidosByUser(req.user.id);
        res.json(pedidos);
    } catch (error) {
```

```
        res.status(500).json({ error: 'Error al obtener pedidos' });
    }
});
```

BASE DE DATOS - TABLAS PRINCIPALES

1. Tabla `usuarios`

```
CREATE TABLE usuarios (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nombre TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    password TEXT NOT NULL,          -- Hash encriptado
    rol TEXT NOT NULL,               -- 'cliente', 'profesional', 'admin'
    categoria TEXT,                  -- Solo profesionales
    aprobado INTEGER DEFAULT 0,      -- 0=pendiente, 1=aprobado
    saldo REAL DEFAULT 0,            -- Solo profesionales
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Ejemplo de registro:

id	nombre	email	rol	aprobado	saldo
1	Admin	admin@hogarfix.local	admin	1	0
2	Juan Pérez	juan@test.com	profesional	1	425
3	María	maria@test.com	cliente	1	0

2. Tabla `pedidos`

```
CREATE TABLE pedidos (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    cliente_id INTEGER NOT NULL,
    profesional_id INTEGER NOT NULL,
    descripcion TEXT NOT NULL,
    precio REAL,
    estado TEXT DEFAULT 'pendiente',  -- pendiente, confirmado, completado
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (cliente_id) REFERENCES usuarios(id),
    FOREIGN KEY (profesional_id) REFERENCES usuarios(id)
);
```

Estados del pedido:

- `pendiente`: Cliente envió solicitud, profesional no ha confirmado

- `confirmado`: Profesional confirmó precio
- `completado`: Servicio terminado, cliente puede calificar

3. Tabla `mensajes`

```
CREATE TABLE mensajes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    pedido_id INTEGER NOT NULL,
    remitente_id INTEGER NOT NULL,
    destinatario_id INTEGER NOT NULL,
    mensaje TEXT,
    imagen TEXT,                    -- URL de imagen (opcional)
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (pedido_id) REFERENCES pedidos(id)
);
```

FUNCIONALIDADES DESTACADAS

1. **Chat en Tiempo Real**

¿Cómo funciona?

```
// El frontend consulta cada 3 segundos si hay mensajes nuevos
setInterval(async () => {
    const mensajes = await fetch(`/api/mensajes/pedido/${pedidoId}`);
    mostrarMensajes(mensajes);
}, 3000);
```

Mejora futura: Usar WebSockets para comunicación instantánea real.

2. **Sistema de Calificaciones**

```
// Cliente califica después de servicio completado
POST /api/ratings
Body: {
    pedidoId: 42,
    estrellas: 5,
    comentario: "¡Excelente trabajo!"
}

// Profesional ve su promedio
GET /api/ratings/profesional/5
Response: {
    promedio: 4.7,
    totalReviews: 23
}
```

3. **Gestión de Saldo Profesional**

```
// Cuando se completa un pedido:
const precio = 500;
const comision = 0.15;  // 15% para la plataforma
const ganancia = precio * (1 - comision);  // 425

UPDATE usuarios SET saldo = saldo + 425 WHERE id = profesional_id;
```

4. **Backup Automático**

```
// Cada 24 horas, crea una copia de seguridad
setInterval(() => {
    const timestamp = new Date().toISOString().replace(/[:.]/g, '-');
    const backupPath = `backups/database_${timestamp}.db`;
    fs.copyFileSync('database.db', backupPath);
    console.log('Backup creado:', backupPath);
}, 24 * 60 * 60 * 1000);  // 24 horas en milisegundos
```

📦 DEPENDENCIAS (package.json)

¿Qué son las dependencias?

Son bibliotecas de código que otros desarrolladores crearon y tú usas para no reinventar la rueda.

Principales dependencias de HogarFix:

```
{
  "express": "^4.18.2",        // Framework del servidor
  "sqlite3": "^5.1.6",         // Base de datos
  "jsonwebtoken": "^9.0.2",    // Tokens de autenticación
  "bcryptjs": "^2.4.3",        // Encriptar contraseñas
  "cors": "^2.8.5",            // Permitir peticiones desde frontend
  "multer": "^1.4.5-lts.1"     // Subir archivos (imágenes del chat)
}
```

¿Cómo se instalan?

```
cd backend
npm install  # Lee package.json e instala todo automáticamente
```

🚀 DESPLIEGUE EN PRODUCCIÓN

¿Qué es "despliegue"?

Es subir tu aplicación a un servidor en internet para que cualquiera pueda acceder (no solo localhost).

Render.com - Plataforma de Hosting

¿Por qué Render?

- Gratis para proyectos pequeños
- Se conecta automáticamente con GitHub
- Cuando haces `git push`, Render actualiza la app automáticamente

Archivo `render.yaml`:

```
services:
  - type: web
    name: hogarfix
    env: node
    buildCommand: npm install --prefix backend
    startCommand: cd backend && npm start
    envVars:
      - key: NODE_ENV
        value: production
```

¿Qué hace?

1. `buildCommand`: Instala dependencias
2. `startCommand`: Inicia el servidor
3. `envVars`: Variables de entorno (configuraciones)

URL de Producción:

```
https://hogarfix.onrender.com
```

🔍 CONSEJOS PARA LA DEFENSA DEL PROYECTO

1. **Explica el Problema que Resuelve**

"Muchas personas necesitan servicios para el hogar pero no saben a quién llamar. HogarFix conecta de forma segura a clientes con profesionales verificados."

2. **Destaca la Arquitectura**

"Usamos una arquitectura de tres capas: frontend (lo que ve el usuario), backend (lógica de negocio) y base de datos (almacenamiento persistente). Esto facilita el mantenimiento y escalabilidad."

3. **Menciona la Seguridad**

"Implementamos autenticación con JWT y encriptación de contraseñas con bcrypt. Además, tenemos un sistema de roles que controla quién puede hacer qué."

4. **Muestra el Flujo Completo**

"Un cliente se registra, busca un profesional por categoría, solicita un servicio, el profesional confirma el precio, se comunican por chat, se completa el trabajo, y el cliente puede calificar."

5. **Habla de Futuras Mejoras**

- Chat en tiempo real con WebSockets
- Pagos online integrados
- App móvil nativa
- Sistema de geolocalización
- Notificaciones push

☑ CHECKLIST ANTES DE LA PRESENTACIÓN

- ☐ Verificar que https://hogarfix.onrender.com funciona
- ☐ Tener cuentas de prueba listas:
  - Admin: admin@hogarfix.local / admin123
  - Cliente: cliente@test.com / 123456
  - Profesional: pro@test.com / 123456
- ☐ Crear datos de prueba (pedidos, mensajes, calificaciones)
- ☐ Practicar el flujo completo
- ☐ Tener laptop con batería completa
- ☐ Llevar cable HDMI por si hay que conectar a proyector
- ☐ Tener screenshots de respaldo por si falla internet

FIN DE LA PARTE 1

Continúa en: `GUIA_TECNICA_PARTE2.md`