Actividad: Proyecto de Graduación

Nombre de la institución

Universidad Internacional San Isidro Labrador

Curso:

Programación Avanzada

Código: ISB-32

Profesor

Estefania Boza Villalobos

Estudiante:

Melanie Sofia Salas Ávila

Nombre del Proyecto:

Aplicación web para Distribuidora Quesada Umaña

Contenido

Índice de Ilustraciones

Objetivo General

Diseñar e implementar una aplicación web para la Distribuidora Quesada Umaña, que permita a los clientes ver e interactuar de una manera más detallada con el catálogo de lámparas que posee dicha empresa y que el dueño de la empresa pueda administrar el catálogo de una manera segura pudiendo agregar, quitar y modificar productos del catálogo.

Objetivos Específicos

- Proporcionar una visualización detallada y atractiva del catálogo de lámparas.
- Proveer un sistema de filtrado eficiente para la búsqueda de productos.
- Facilitar la administración del catálogo para el dueño de la distribuidora.
- Implementar un sistema de autenticación para que solo el dueño pueda administrar el catálogo.
- Implementar una base de datos para guardar todos los productos ingresados.

Justificación

La compra de lámparas suele ser más efectiva cuando se realiza de manera presencial, pero en la actualidad los clientes no tienen suficiente tiempo. Por este motivo, buscan una experiencia de compra eficaz, rápida y agradable, necesitan visualizar detalladamente el producto que desean obtener, pero de manera virtual. Una página web para la Distribuidora Quesada Umaña mejorara esa experiencia de elección de productos para el cliente y al mismo tiempo mejorando la eficiencia de las ventas de las lámparas. Al mismo tiempo, ofreciéndole al dueño una administración del catálogo segura y fácil de manejar, manteniendo siempre actualizado el inventario de productos disponibles.

Alcances del proyecto

- Catálogo de lámparas: La aplicación web debe permitir ver las lámparas disponibles con detalles como fotos, descripciones, especificaciones técnicas y precios.
- Búsqueda y filtrado: La aplicación web de incluir un sistema de búsqueda y filtrado para encontrar los productos por diferentes categorías como tipo, tamaño, precio, entre otros.

- Gestión del catálogo: Proveer una interfaz sencilla y segura para que el dueño pueda agregar, quitar y modificar productos del catálogo.
- Autenticación del administrador: crear un sistema de autenticación exclusivo par que solo el dueño pueda manipular el catálogo.
- Base de datos: implementar una base de datos para respaldar el catalogo con todos los productos ingresados por el dueño.
- Sección de reacciones y comentarios: Permitir que los clientes puedan dejar reacciones y comentarios sobre los productos, proporcionando retroalimentación valiosa para mejorar la oferta y aumentar la confianza en las compras.

Requerimientos

Requerimientos funcionales:

- La aplicación debe proporcionarles a los usuarios explorar el catálogo de lámparas con todos los detalles necesarios.
- Implementar una barra de búsqueda y filtros avanzados para facilitar la localización de los productos.
- Crear un sistema de autenticación para el dueño para la gestión del catálogo.
- Una interfaz agradable, sencilla para la administración donde el dueño pueda realizar las modificaciones al catálogo.
- Implementar una base de datos robusta para almacenar todos los datos necesarios del catálogo.
- Integrar una sección donde los clientes puedan dejar comentarios y reacciones sobre los productos, ayudando a generar confianza y mejorar la experiencia de compra.

Requerimientos no funcionales:

- La interfaz debe ser atractiva para los clientes y adaptable para cualquier dispositivo, ya sean computadoras o dispositivos móviles.
- La aplicación debe seguir buenas prácticas de seguridad para proteger los datos del administrador y la información del catálogo.
- El rendimiento debe de ser óptimo para manejar un catálogo extenso.
- Base de datos que soporte la extensa carga de información de los productos y un respaldo de la información.

- La sección de comentarios y reacciones debe ser moderada para garantizar un ambiente adecuado y útil para los clientes.

Diseño del frontend de TaskWise

Stack y herramientas principales

- React + Vite: aplicación SPA creada con Vite (arranque rápido, HMR) y React (componentes funcionales).
- React Router DOM: enrutamiento de páginas; se usa BrowserRouter en main.jsx.
- Context API (AuthContext): manejo centralizado de autenticación (usuario y token), con AuthProvider que envuelve a <App />.
- Axios: cliente HTTP centralizado en src/services/api.js con:
    - baseURL por entorno (VITE_API_URL en .env).
    - Persistencia de accessToken en localStorage.
    - Helper setAuthToken(token) para inyectar/eliminar Authorization: Bearer.
- Tailwind CSS: estilos utilitarios; darkMode: 'class' + plugin de formularios/typography/line-clamp.
- UI y UX: iconos con lucide-react, animaciones con framer-motion, notificaciones con react-toastify.

Estructura de carpetas relevante (src/)

- main.jsx: monta la app con <BrowserRouter> y <AuthProvider>.
- App.jsx: define el armazón de rutas públicas vs protegidas (usa Suspense y patrón de rutas privadas).
- routes/PrivateRoute.jsx: guard de rutas: si no hay user, redirige a /login.
- context/AuthContext.jsx:
    - Restaura sesión leyendo accessToken/auth_user de localStorage.
    - Expone { user, token, login, logout }.
    - login guarda token/usuario y sincroniza Authorization; logout limpia todo.
- hooks/: useAuth.js (consumo del contexto), useDarkMode.js, useTasks.js.

- services/:
    - api.js (instancia axios + setAuthToken).
    - projectService.js, taskService.js, roleService.js con funciones CRUD.
    - Normalizadores de respuesta para tolerar distintos formatos de backend (p.ej. {data, meta}, {rows, count}, items+pagination).
- api/authService.js: loginUser/registerUser con normalización flexible de payloads de auth (acepta accessToken|token|jwt, user|profile|userData, etc.).
- layout/: Navbar.jsx, Sidebar.jsx, Footer.jsx y Layout.jsx (estructura común de páginas autenticadas).
- components/:
    - Formularios y UI: LoginForm.jsx, RegisterForm.jsx, AddTaskForm.jsx, CreateProjectForm.jsx, Input.jsx, Button.jsx, ConfirmDialog.jsx, Spinner.jsx, DarkModeSwitch.jsx.
    - Tablas/listas: TaskList.jsx.
    - Métricas: DashboardCard.jsx.
- pages/:
    - Públicas: Landing.jsx, Login.jsx.
    - Protegidas: Dashboard.jsx, Proyectos.jsx, ProjectTasksPage.jsx, Tareas.jsx, Reportes.jsx.
    - Admin: RoleManager.jsx, PriorityTypes.jsx, StatusManager.jsx, TagManager.jsx.
    - Otras: Soporte.jsx, Profile.jsx, Settings.jsx.

Flujo de autenticación (front)

1. El usuario inicia sesión en /login (página + LoginForm.jsx).
2. loginUser(credentials) llama POST /auth/login.
3. La respuesta se normaliza (acepta variaciones de nombres de campos).
4. Se guarda accessToken y usuario en localStorage y se fija Authorization en Axios mediante setAuthToken.
5. Las rutas protegidas se habilitan; PrivateRoute asegura que sin user → redirect a /login.
6. En recargas, AuthContext rehidrata sesión leyendo localStorage.

Navegación y layout

- Layout autenticado: Layout.jsx compone Navbar + Sidebar + main (tema claro/oscuro por clase dark en <html>; toggle con DarkModeSwitch).
- Navbar/Sidebar abren secciones según la URL (p. ej. /proyectos, /tasks, /roles, /priorities, /statuses, /reportes, /profile, /settings, /soporte).
- Rutas privadas envueltas por PrivateRoute (ej.: Dashboard, Proyectos, Tareas, Admin…).
- Rutas públicas: Landing y Login.

Gestión de datos y formularios

- Servicios tipados por dominio (projectService, taskService, roleService) aíslan llamadas HTTP del resto de la UI.
- Formularios reusables (Input, Button) y específicos de dominio (AddTaskForm, CreateProjectForm).
- ConfirmDialog para acciones destructivas; Toast para feedback no intrusivo.
- Listados (TaskList) con estados de carga/vacío/errores.
- Dashboard usa tarjetas (DashboardCard) para KPIs rápidas.

Tematización y estilos

- Tailwind con plugins (forms/typography/line-clamp).
- Dark mode por clase: el switch agrega/remueve .dark (config en tailwind.config.js).
- Componentes desacoplados facilitan cambiar estilo sin reescribir lógica.

Configuración, build y entorno

- Vite (vite.config.js) con plugin React.
- Variables de entorno: .env expone VITE_API_URL para apuntar al backend por ambiente.
- Eslint configurado; estructura lista para producción (build con vite build).

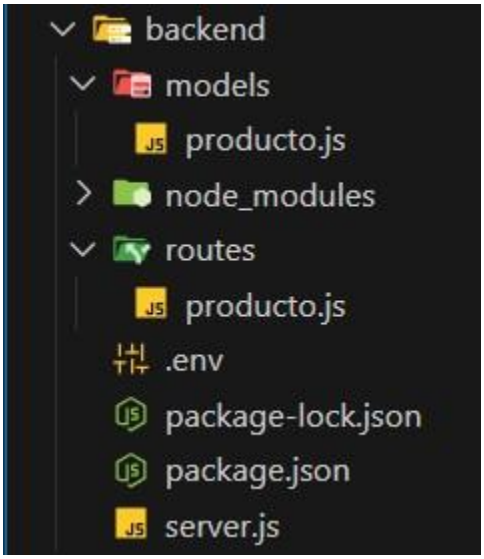
Consideraciones de seguridad del lado del front

- Token en localStorage (simple y común, aunque vulnerable a XSS si existiera).
- Guard de rutas en front evita mostrar UI sensible; el backend debe reforzar autorización en las API.
- setAuthToken centraliza la cabecera Authorization para todas las peticiones.

Tecnologías utilizadas

Durante el desarrollo del proyecto, tuve que investigar y probar con diferentes tecnologías, toda la investigación, análisis, aprendizaje y pruebas fue para encontrar la más funcional y práctica para mí. El motor principal para esto es la pila de desarrollo web MEAN que usa MongoDB, Express.js, Angular y Node.js donde el Frontend está con Angular, el Backend es con Node.js y Express.js y mi base de datos MongoDB que no es relacional y el lenguaje que usé fue TypeScript. Estas son tecnologías que nunca antes había usado y tuve que aprender a usarlas desde cero, pero también puedes crear proyectos más organizados y eficientes. Para administrar mi base de datos, creé una API usando Node.js, Express.js y MongoDB para administrar mi tabla en la base de datos y la función CRUD principal del gerente de producto.

Ilustración 1: Estructura del backend



Este es mi estructura del backend API, donde tengo el manejo y las funciones del CRUD y además las rutas de conexión con mi base de datos.

- En la carpeta models, tengo el archivo donde le muestro el formato que quiero para la tabla de mi base de datos.
- En routes está la comunicación entre el Frontend y el Backend, es el intermediario que conecta mi página web con la base de datos, donde también le envía las solicitudes de crear, leer, actualizar y eliminar y este mismo las procesa y las aplica a mi base de datos en
- MongoDB.

Justificación técnica

Para el desarrollo de este proyecto se eligió el stack MEAN, compuesto por MongoDB, Express.js,

Angular y Node.js, porque permite crear una aplicación completa utilizando un solo lenguaje, JavaScript/TypeScript, en todas las capas. Angular se utilizó en el frontend por su facilidad para construir interfaces dinámicas y reactivas, mientras que Node.js y Express.js en el backend facilitan manejar solicitudes y crear una API que comunica la interfaz con la base de datos.

MongoDB se eligió por ser flexible y permitir almacenar los productos en formato de documentos JSON, lo que hace más sencilla la gestión de datos sin estructuras rígidas. Además, librerías como

Mongoose ayudan a organizar los datos y ejecutar operaciones CRUD de forma más eficiente, y Bootstrap con sus íconos aporta un diseño profesional y responsive sin complicaciones. En conjunto, estas tecnologías permiten que el proyecto sea funcional, organizado, escalable y fácil de mantener, aun cuando fueron herramientas que se aprendieron desde cero durante el desarrollo.

Diagrama de la base de datos

Esta es la estructura de mi base de datos creada en MongoDB, la base de datos se llama catálogo, con una única tabla llamada productos, este proyecto es un prototipo y se espera poder desarrollarlo más para que pueda ser utilizado realmente por la distribuidora Quesada Umaña, este proyecto no tiene base de datos relacional y solo pose una tabla ya que cree un API para que me administrara la tabla y también me creara los campos de la tabla desde el

backend del proyecto. Toda la administración de información de la tabla esta administrada por la API. A continuación, adjuntare screenshots de mi proyecto, la base de datos y la tabla.

Ilustración 2: Diagrama entidad relación

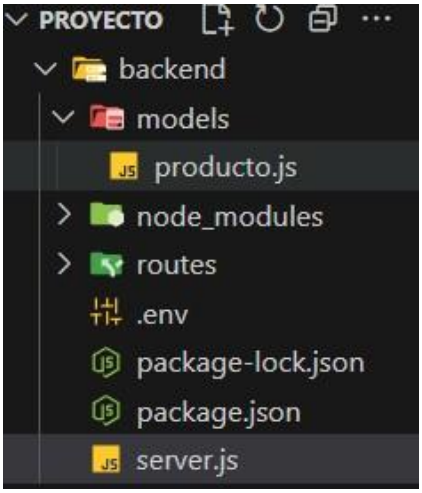
Ilustración 3: Campos de la tabla

La segunda imagen muestra los campos que tiene la tabla productos, los campos id, nombre, descripción y la imagen URL que la imagen es almacenada en Firebase que lo que hace es convertir la imagen a una URL y me la almacena en la base de datos.

Requerimientos implementados

Adjunto capturas de la API y de la estructura del backend,

Ilustración 4: Carpeta models



En la carpeta de models se encuentra producto j.s que es donde le damos los requerimientos y la estructura que queremos que tenga la tabla llamando a una librería de mongo para que me llame a mi base de datos.

Ilustración 5: Envió de datos a table producto

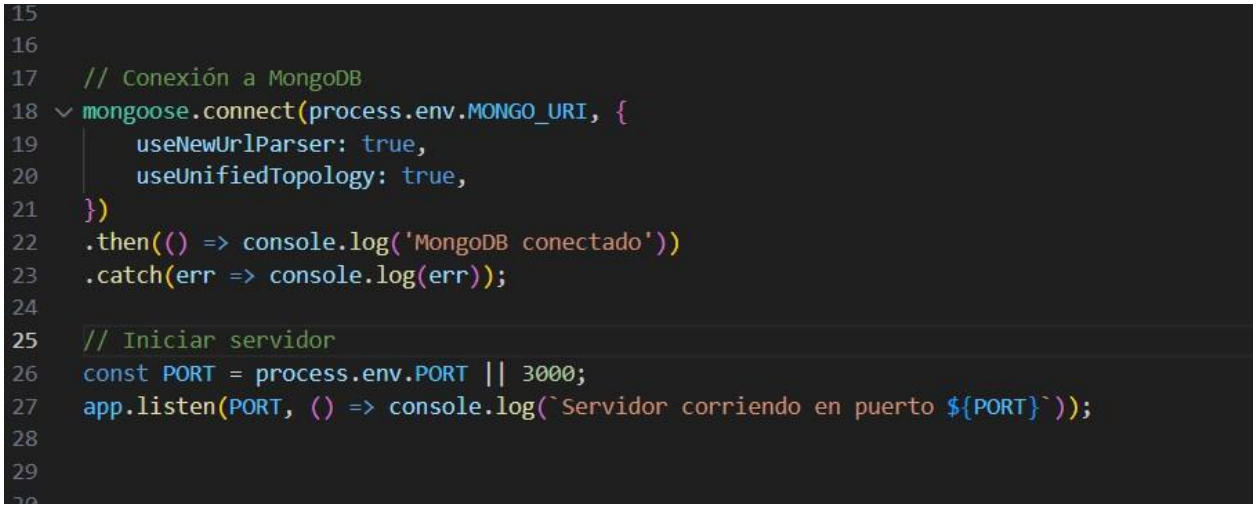
```
producto.js
backend > models > producto.js > ...
1  const mongoose = require('mongoose');
2
3  const ProductoSchema = new mongoose.Schema({
4      nombre: { type: String, required: true },
5      descripcion: { type: String, required: true },
6      imagenURL: { type: String, required: true },
7  }, { timestamps: true });
8
9  module.exports = mongoose.model('Producto', ProductoSchema);
```

Código de conexión a la base de datos

Explicación breve del código de conexión a la base de datos en MongoDB.

Ilustración 6: Conexión con la base: Imágenes de sistema

```
server.js
backend > server.js > ...
1  const express = require('express');
2  const mongoose = require('mongoose');
3  const cors = require('cors'); // permite que reciban solicitudes desde el frontend
4  require('dotenv').config();
5
6  const app = express();
7
8  // Middlewares
9  app.use(cors());
10 app.use(express.json());  //permita que entienda el json de las solicitudes
11
12 // Rutas
13 const productoRoutes = require('./routes/producto');
14 app.use('/api/producto', productoRoutes); // las rutas que estan definidas por producRou estan bajo api
15
```



```
// Conexión a MongoDB
mongoose.connect(process.env.MONGO_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
})
.then(() => console.log('MongoDB conectado'))
.catch(err => console.log(err));

// Iniciar servidor
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Servidor corriendo en puerto ${PORT}`));
```

Cada sección del código tiene un comentario con una breve explicación de para que sirve el código, básicamente todo el código nos permite acceder a las rutas donde podremos conectarlas al frontend, que pueda entender el Json de las solicitudes y que pueda iniciar el servidor de la base de datos.

Conclusión del programa *TaskWise*

El desarrollo de TaskWise ha culminado exitosamente, integrando de manera sólida y eficiente tanto el backend como el frontend. En el backend, se implementó una arquitectura robusta basada en Node.js con Express, conectada a una base de datos MongoDB. Esta estructura permite gestionar la información mediante una API REST que soporta operaciones CRUD, asegurando una administración completa de los productos, usuarios, proyectos y tareas.

Por su parte, el frontend, construido con Angular, ofrece una interfaz intuitiva y funcional. Componentes clave como *Home* y *Productos* permiten al usuario navegar fácilmente por el sistema y realizar acciones sobre los datos, gracias a la integración directa con los servicios del backend. Esta comunicación fluida mediante HTTP garantiza la sincronización constante entre la interfaz y la base de datos.

TaskWise destaca por su arquitectura modular, que no solo facilita el mantenimiento y la escalabilidad, sino que también abre la puerta a futuras integraciones como notificaciones, calendarios externos o aplicaciones móviles. Con funcionalidades orientadas a la productividad y colaboración —como la gestión de usuarios, proyectos, tareas y prioridades— TaskWise se consolida como una herramienta versátil y adaptable a diversos entornos profesionales.

Vídeo del sistema funciona

https://drive.google.com/drive/folders/1Kj8KeK962B6vpZhbtH80huogggmIdfyV?usp=sharing