

INTRODUCCIÓN

En el contexto de sistemas colaborativos modernos, la necesidad de comunicación en tiempo real se ha vuelto indispensable. La sincronización inmediata entre usuarios permite mejorar significativamente la experiencia de uso, agilizar los flujos de trabajo y mantener la información siempre actualizada sin necesidad de recargar la interfaz.

Para lograr esto, el sistema *task_manager* implementa WebSockets mediante la biblioteca Socket.IO, que permite establecer una conexión bidireccional persistente entre el cliente y el servidor. A diferencia del modelo tradicional basado exclusivamente en peticiones HTTP, los WebSockets permiten que el servidor envíe información de manera proactiva al cliente sin necesidad de que este la solicite continuamente (polling), logrando así un canal de comunicación más eficiente y reactivo.

Mediante Socket.IO se implementan múltiples eventos personalizados, organizados según su dirección: del cliente al servidor y del servidor al cliente. Estos eventos permiten operaciones como unirse a proyectos, recibir notificaciones de tareas nuevas o actualizadas, detectar usuarios conectados y gestionar comentarios en tiempo real. Esto resulta especialmente útil en entornos colaborativos, donde varios usuarios pueden estar trabajando simultáneamente sobre los mismos recursos del sistema.

OBJETIVOS DEL MANUAL

Objetivo general

Proveer una guía técnica para facilitar la implementación, capacitación y mantenimiento del sistema.

Objetivos específicos

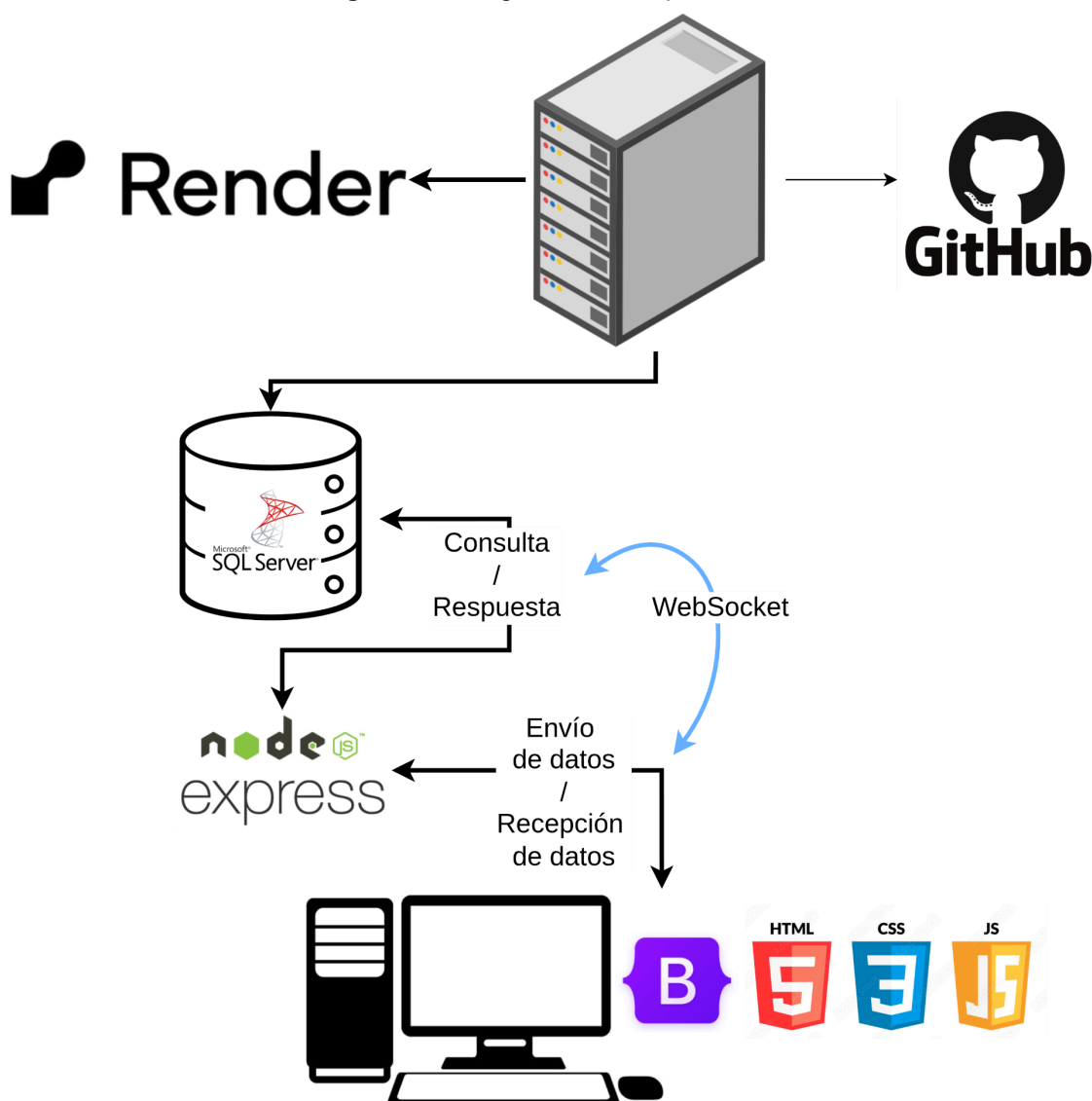
- Ofrecer una guía técnica que detalle la arquitectura, base de datos, procesos y aspectos clave para la comprensión efectiva y sin problemas.
- Brindar instrucciones claras de la instalación y configuración inicial del sistema, así como la configuración de un entorno de desarrollo.

CAPÍTULO 1: DISEÑO DEL SISTEMA

El diseño del sistema constituye una fase fundamental en el desarrollo de software, ya que define la estructura técnica, arquitectónica y tecnológica que permite transformar los requisitos funcionales en una solución práctica y eficiente. En este capítulo se detallan los componentes, herramientas y tecnologías que componen la arquitectura del sistema tal y como se muestra en la figura 1.1, así como las decisiones de diseño que aseguran su correcto funcionamiento, escalabilidad y mantenibilidad a largo plazo.

El sistema propuesto sigue una arquitectura moderna basada en servicios, que permite una comunicación fluida entre el cliente y el servidor mediante API REST y WebSockets, facilitando así la interacción en tiempo real y el acceso seguro a los datos. A continuación, se describen las principales tecnologías utilizadas y se justifica su elección.

Figura 1.1: Diagrama de componentes.



Fuente: Los autores, con la herramienta draw.io.

1.1 Tecnologías utilizadas

1.1.1 GitHub – Control de versiones

GitHub es una plataforma basada en Git utilizada para el control de versiones del código fuente. Permite trabajar de forma colaborativa, mantener un historial claro de cambios, gestionar ramas de desarrollo y facilitar la integración continua. Su uso garantiza trazabilidad, respaldo y organización del trabajo en equipo durante todo el ciclo de vida del proyecto.

1.1.2 Render – Plataforma de despliegue en la nube

Render es una plataforma de alojamiento en la nube que permite desplegar aplicaciones web y servicios backend de manera automática a partir de un repositorio en GitHub. Su simplicidad de configuración y su integración directa con Git hacen posible un flujo de trabajo continuo (CI/CD), lo cual facilita actualizaciones frecuentes y un entorno de producción estable.

1.1.3 SQL Server – Sistema Gestor de Base de Datos (DBMS)

Se utilizó Microsoft SQL Server como sistema de gestión de base de datos debido a su robustez, soporte transaccional, integridad referencial y capacidad de escalar con eficiencia. Es especialmente adecuado para aplicaciones con estructuras de datos complejas, relaciones bien definidas y necesidad de auditoría y seguridad en el manejo de la información.

1.1.4 Node.js y Express – Backend del sistema

El backend del sistema fue desarrollado con Node.js, un entorno de ejecución de JavaScript en el servidor, y el framework Express, que facilita la creación de rutas, controladores y middleware. Esta combinación permitió implementar una API REST robusta y endpoints WebSocket para funcionalidades en tiempo real, como notificaciones o actualizaciones dinámicas del sistema.

1.1.5 HTML, CSS y Bootstrap – Interfaz de usuario (Frontend)

La interfaz del sistema se construyó utilizando HTML5 y CSS3, complementados con Bootstrap, un framework de diseño responsive que permite crear interfaces modernas, limpias y adaptables a distintos dispositivos. Esta tecnología asegura una experiencia de usuario amigable, accesible y coherente visualmente.

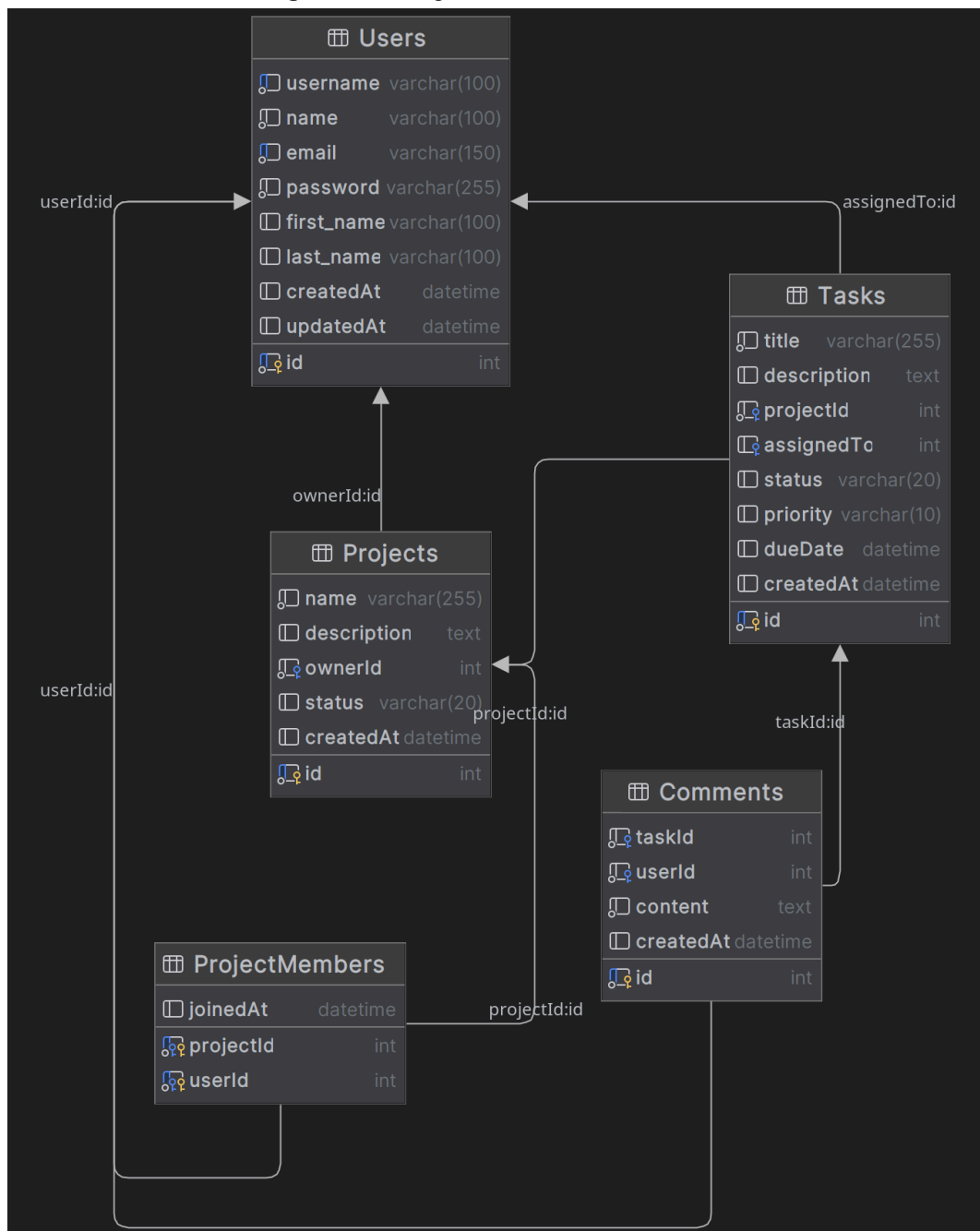
El conjunto de tecnologías seleccionadas responde tanto a criterios técnicos como estratégicos. Entre los beneficios más importantes se destacan:

- **Modularidad y escalabilidad:** gracias a la separación entre frontend, backend y base de datos.
- **Desarrollo ágil y mantenible:** con el uso de GitHub y buenas prácticas de control de versiones.
- **Despliegue automatizado y continuo:** mediante Render, lo cual reduce los errores humanos y agiliza la entrega.
- **Comunicación en tiempo real:** posible gracias al uso de WebSockets implementados en Node.js.
- **Compatibilidad y estandarización:** mediante el uso de tecnologías ampliamente adoptadas y bien documentadas.

CAPÍTULO 2: BASE DE DATOS Y DICCIONARIO DE DATOS

El diseño de la base de datos (figura 2.1) es un componente esencial en el desarrollo de cualquier sistema de información, ya que constituye la estructura central donde se almacenan, organizan y gestionan los datos necesarios para el funcionamiento del sistema. En el caso del presente proyecto, la base de datos denominada *task_manager* ha sido diseñada con un enfoque relacional utilizando Microsoft SQL Server como sistema de gestión de base de datos (DBMS), garantizando así integridad, consistencia y escalabilidad.

Figura 2.1: Diagrama de la base de datos.



Fuente: Los autores, con la herramienta DataGrip-JetBrains.

La base de datos está compuesta por un conjunto de tablas interrelacionadas que reflejan las entidades y relaciones principales del sistema, como usuarios, proyectos, tareas, comentarios y miembros de proyectos. Cada tabla ha sido cuidadosamente estructurada con tipos de datos adecuados, claves primarias y foráneas, restricciones y valores por defecto, lo cual facilita la validación automática y el control de calidad de los datos desde la capa de persistencia. Uno de los principios fundamentales en el diseño fue aplicar un modelo normalizado, lo que permitió reducir la redundancia, mejorar la coherencia de los datos y optimizar las operaciones de consulta, inserción y actualización. Esta normalización asegura que cada dato se almacene una sola vez en el lugar correspondiente, reduciendo así el riesgo de inconsistencias lógicas en la información.

2.1 Diccionario de datos

El diccionario de datos cumple un rol esencial como recurso de consulta, que facilita la identificación y comprensión de las entidades y atributos presentes en las tablas de la base de datos. A continuación se describe cada una de las tablas que conforman la base de datos.

2.1.1 Tabla Users

Tabla 2.1: Diccionario de datos tabla Users.

Campo	Tipo	Restricciones	Descripción
id	INT	PK, AUTO_INCREMENT	Identificador único del usuario.
username	VARCHAR(100)	NOT NULL, UNIQUE	Nombre de usuario único para inicio de sesión.
name	VARCHAR(100)	NOT NULL	Nombre completo del usuario.
email	VARCHAR(150)	NOT NULL, UNIQUE	Correo electrónico único del usuario.
password	VARCHAR(255)	NOT NULL	Contraseña encriptada del usuario.
first_name	VARCHAR(100)	NULL	Primer nombre del usuario (opcional).
last_name	VARCHAR(100)	NULL	Apellido del usuario (opcional).
createdAt	DATETIME	DEFAULT GETDATE()	Fecha de creación del registro.
updatedAt	DATETIME	DEFAULT GETDATE()	Fecha de última actualización del registro.

Fuente: Los autores.

2.1.2 Tabla Projects

Tabla 2.2: Diccionario de datos tabla Projects.

Campo	Tipo	Restricciones	Descripción
id	INT	PK, AUTO_INCREMENT	Identificador único del proyecto.
name	VARCHAR(255)	NOT NULL	Nombre del proyecto.
description	TEXT	NULL	Descripción del proyecto.
ownerId	INT	NOT NULL, FK → Users(id)	ID del usuario que creó el proyecto.
status	VARCHAR(20)	DEFAULT 'Activo', CHECK	Estado del proyecto: 'Activo' o 'Completado'.
createdAt	DATETIME	DEFAULT GETDATE()	Fecha de creación del proyecto.

Fuente: Los autores.

2.1.3 Tabla ProjectMembers

Tabla 2.3: Diccionario de datos tabla ProjectMembers.

Campo	Tipo	Restricciones	Descripción
projectId	INT	PK, FK → Projects(id)	ID del proyecto.
userId	INT	PK, FK → Users(id)	ID del usuario que es miembro del proyecto.
joinedAt	DATETIME	DEFAULT GETDATE()	Fecha de incorporación al proyecto.

Fuente: Los autores.

2.1.4 Tabla Tasks

Tabla 2.4: Diccionario de datos tabla Tasks.

Campo	Tipo	Restricciones	Descripción
id	INT	PK, AUTO_INCREMENT	Identificador único de la tarea.
title	VARCHAR(255)	NOT NULL	Título breve de la tarea.
description	TEXT	NULL	Descripción detallada de la tarea.

projectId	INT	NOT NULL, FK → Projects(id)	Proyecto al que pertenece la tarea.
assignedTo	INT	FK → Users(id)	Usuario asignado a la tarea (puede ser nulo).
status	VARCHAR(20)	DEFAULT 'Por hacer', CHECK	Estado: 'Por hacer', 'En progreso', 'Completado'.
priority	VARCHAR(10)	DEFAULT 'Media', CHECK	Prioridad: 'Baja', 'Media', 'Alta'.
dueDate	DATETIME	NULL	Fecha límite para la tarea.
createdAt	DATETIME	DEFAULT GETDATE()	Fecha de creación de la tarea.

Fuente: Los autores.

2.1.5 Tabla Comments

Tabla 2.5: Diccionario de datos tabla Comments.

Campo	Tipo	Restricciones	Descripción
id	INT	PK, AUTO_INCREMENT	Identificador único del comentario.
taskId	INT	NOT NULL, FK → Tasks(id)	Tarea a la que pertenece el comentario.
userId	INT	NOT NULL, FK → Users(id)	Usuario que hizo el comentario.
content	TEXT	NOT NULL	Contenido del comentario.
createdAt	DATETIME	DEFAULT GETDATE()	Fecha en que se hizo el comentario.

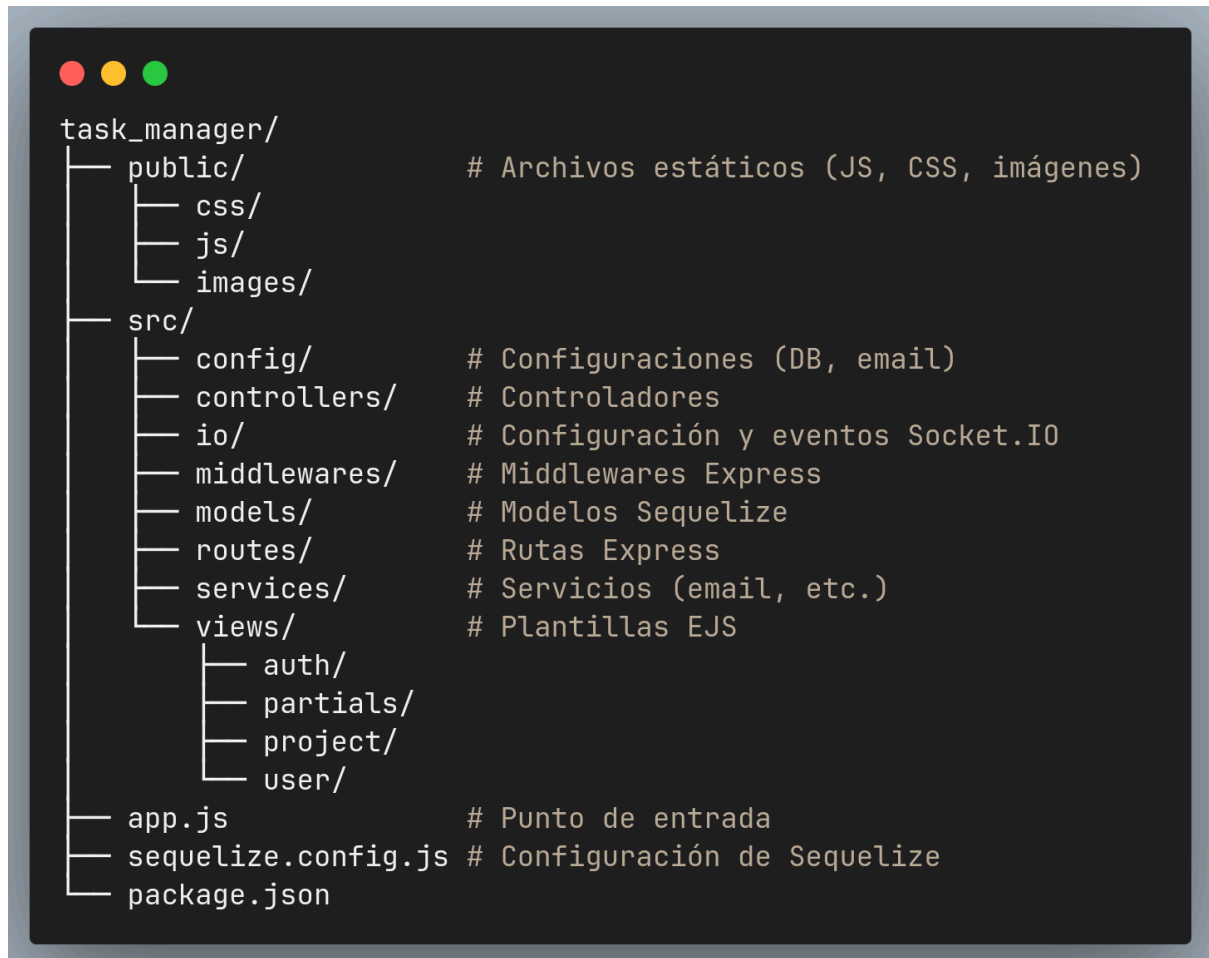
Fuente: Los autores.

La elección de Microsoft SQL Server como sistema de gestión permite aprovechar sus capacidades de manejo transaccional, seguridad y escalabilidad, lo cual es clave para garantizar un buen rendimiento en ambientes de producción. Este diseño proporciona una base sólida para el funcionamiento del sistema, y también facilita futuras integraciones, mantenimientos y ampliaciones del sistema.

CAPÍTULO 3: ESTRUCTURA DE CARPETAS DEL PROYECTO

En la figura 3.1 se muestra como la organización del código fuente en un proyecto es clave para asegurar la mantenibilidad, escalabilidad y comprensión del sistema a lo largo del tiempo. En el sistema *task_manager*, se ha seguido una estructura modular basada en buenas prácticas del ecosistema Node.js, que permite separar claramente las responsabilidades del servidor, el cliente, los modelos de datos, los controladores y las vistas.

Figura 3.1: Estructura del proyecto.



Fuente: Los autores.

A continuación, se describe el propósito de cada uno de los directorios y archivos principales en la estructura del proyecto.

Tabla 3.1: Estructura del proyecto y definición de directorios.

Ruta / Carpeta	Descripción
public/	Contiene archivos estáticos accesibles por el cliente (JS, CSS, imágenes).
css/	Hojas de estilo personalizadas.

js/	Scripts del lado del cliente.
images/	Recursos gráficos como íconos o logos.
src/	Directorio principal del código fuente de la aplicación.
config/	Archivos de configuración (conexión a base de datos, servicios externos).
controllers/	Funciones que manejan la lógica de cada ruta (controladores).
io/	Configuración y eventos de Socket.IO para comunicación en tiempo real.
middlewares/	Funciones intermedias para validaciones, autenticación, etc.
models/	Definición de los modelos de datos con Sequelize (ORM).
routes/	Definición de rutas del sistema organizadas por módulo.
services/	Lógica auxiliar como envío de correos, notificaciones, etc.
views/	Plantillas de interfaz de usuario renderizadas por el servidor (EJS).
auth/	Vistas relacionadas al acceso de usuarios (login, registro).
partials/	Fragmentos comunes como menús, encabezados, pies de página.
project/	Vistas relacionadas con proyectos y tareas.
user/	Vistas relacionadas con el perfil y gestión de usuarios.
app.js	Archivo principal del servidor; inicializa Express, middlewares y rutas.
sequelize.config.js	Configuración específica de Sequelize para conectar con la base de datos.

package.json	Archivo de configuración del proyecto Node.js: dependencias, scripts, etc.
--------------	--

Fuente: Los autores.

CAPÍTULO 4: PREPARAR ENTORNO DE DESARROLLO

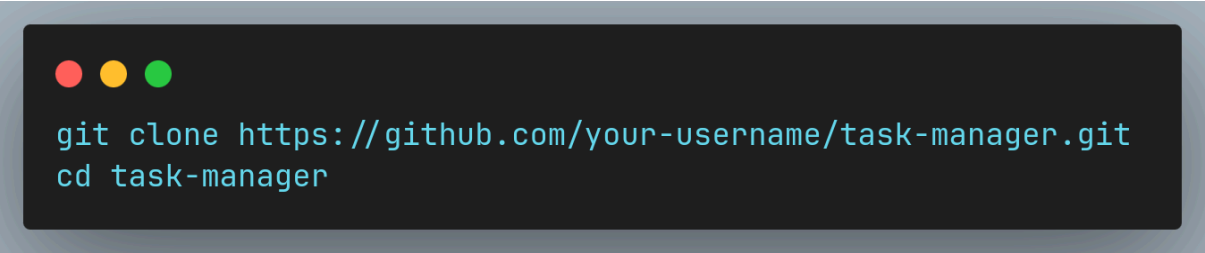
4.1 Requisitos

- Node.js (v14+)
- Microsoft SQL Server (o SQL Server Express).
- Cliente ODBC “ODBC Driver 17 for SQL Server”.
- Git.

4.2 Pasos para preparar entorno de desarrollo

1) Clonar repositorio

Figura 4.1: Comando ejecutar clonar el repositorio.

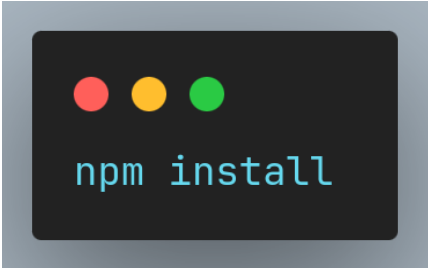


```
git clone https://github.com/your-username/task-manager.git
cd task-manager
```

Fuente: Los autores.

2) Instalar dependencias

Figura 4.2: Comando instalar dependencias.

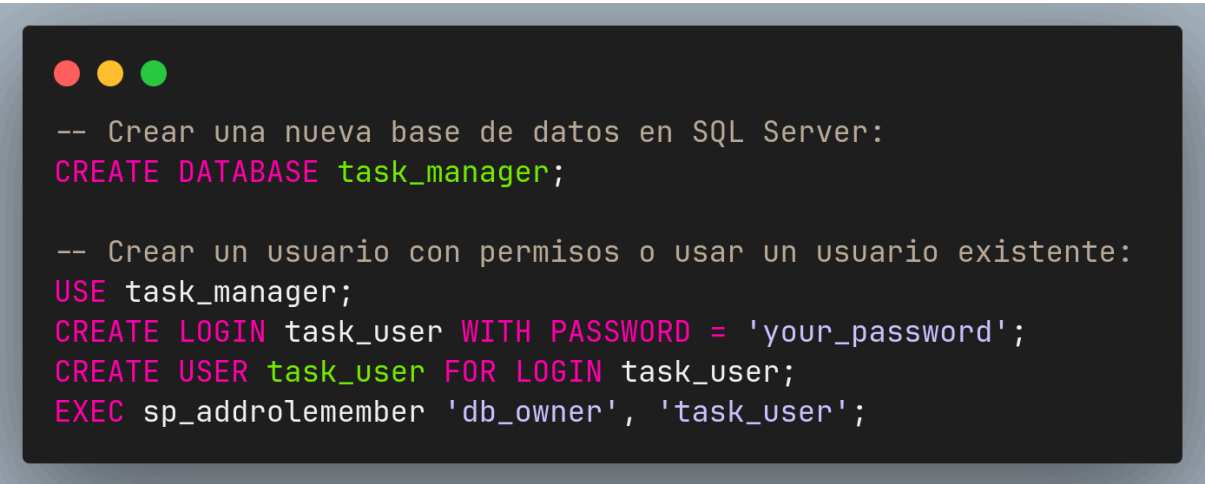


```
npm install
```

Fuente: Los autores.

3) Configurar la base de datos

Figura 4.3: Creación y configuración de la DB.



```
-- Crear una nueva base de datos en SQL Server:
CREATE DATABASE task_manager;

-- Crear un usuario con permisos o usar un usuario existente:
USE task_manager;
CREATE LOGIN task_user WITH PASSWORD = 'your_password';
CREATE USER task_user FOR LOGIN task_user;
EXEC sp_addrolemember 'db_owner', 'task_user';
```

Fuente: Los autores.

4) Configurar variables de entorno

Figura 4.4: Variables de entorno.



```
# Copia el archivo .env.example a .env:
cp .env.example .env

# Edita el archivo .env con tus configuraciones:
# Server
PORT=3000
NODE_ENV=development

# Database (MSSQL)
DB_HOST=your_server_name_or_ip
DB_PORT=1433
DB_NAME=task_manager
DB_USER=task_user
DB_PASS=your_password
DB_DIALECT=mssql

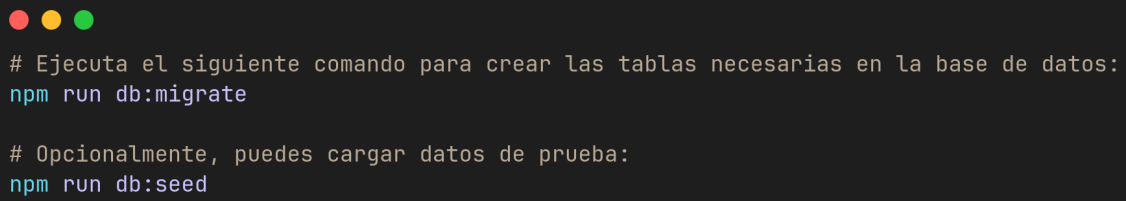
# Session
SESSION_SECRET=generate_a_random_string_here

# Email
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_SECURE=false
EMAIL_USER=your_email@gmail.com
EMAIL_PASS=your_app_password
APP_URL=http://localhost:3000
```

Fuente: Los autores.

5) Ejecutar migraciones de base de datos

Figura 4.5: Migraciones.

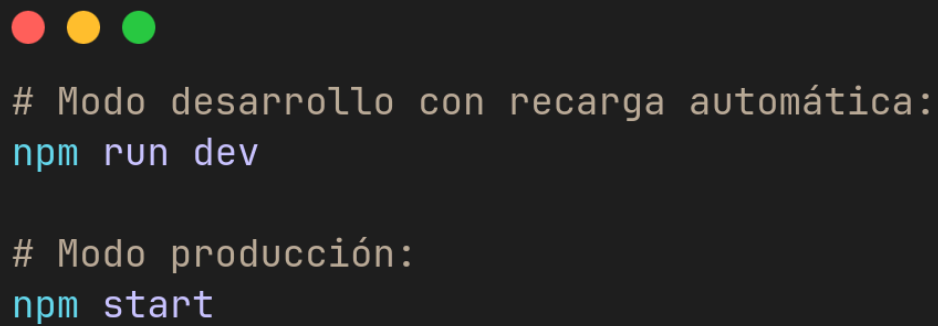


```
# Ejecuta el siguiente comando para crear las tablas necesarias en la base de datos:  
npm run db:migrate  
  
# Opcionalmente, puedes cargar datos de prueba:  
npm run db:seed
```

Fuente: Los autores.

6) Iniciar la aplicación

Figura 4.6: Ejecutar la app.



```
# Modo desarrollo con recarga automática:  
npm run dev  
  
# Modo producción:  
npm start
```

Fuente: Los autores.

CAPÍTULO 5: ENDPOINTS DE LA API

La arquitectura del sistema está respaldada por una API RESTful desarrollada con Node.js y Express, la cual permite gestionar todas las operaciones relacionadas con usuarios, proyectos, tareas y comentarios. Esta API actúa como puente entre el cliente y la base de datos, permitiendo la interacción fluida entre la interfaz de usuario y la lógica del servidor. Los endpoints se encuentran organizados en diferentes módulos funcionales, lo que facilita su mantenimiento, escalabilidad y comprensión. A continuación, se describen los principales endpoints disponibles, agrupados por tipo de operación: autenticación, gestión de proyectos, tareas y comentarios.

Tabla 5.1: Endpoint para autenticación.

Ruta	Método	Descripción
/auth/register	POST	Registrar un nuevo usuario.
/auth/login	POST	Iniciar sesión.
/auth/logout	POST	Cerrar sesión del usuario.

Fuente: Los autores.

Tabla 5.2: Endpoint para proyectos.

Ruta	Método	Descripción
/projects	GET	Listar todos los proyectos del usuario.
/projects/:id/view	GET	Ver detalles de un proyecto específico.
/projects/api/list	GET	API para obtener lista de proyectos.
/projects/api/:id	GET	API para obtener los datos de un proyecto.
/projects	POST	Crear un nuevo proyecto.
/projects/:id/invite	POST	Invitar a un usuario a un proyecto.

Fuente: Los autores.

Tabla 5.3: Endpoint para tareas.

Ruta	Método	Descripción
/projects/:projectId/tasks	GET	Listar tareas asociadas a un proyecto.
/projects/:projectId/tasks	POST	Crear una nueva tarea para un proyecto.
/tasks/:id	GET	Obtener los detalles de una tarea específica.
/tasks/:id	PUT	Actualizar todos los campos de una tarea.
/tasks/:id/status	PATCH	Actualizar solo el estado de una tarea.

/tasks/:id	DELETE	Eliminar una tarea.
------------	--------	---------------------

Fuente: Los autores.

Tabla 5.4: Endpoint para comentarios.

Ruta	Método	Descripción
/tasks/:id/comments	GET	Obtener comentarios asociados a una tarea.
/tasks/:id/comments	POST	Añadir un nuevo comentario a una tarea.

Fuente: Los autores.

CAPÍTULO 6: EVENTOS SOCKET.IO

Para lograr una experiencia en tiempo real en el sistema, se utiliza Socket.IO, una biblioteca que permite establecer comunicación bidireccional entre el cliente y el servidor a través de WebSockets. Esta tecnología permite que múltiples usuarios interactúen simultáneamente con el sistema sin necesidad de recargar la página, recibiendo actualizaciones instantáneas sobre tareas, comentarios y presencia en línea.

Tabla 6.1: Evento Cliente → Servidor.

Evento	Descripción
join-project	El cliente se une a una sala específica del proyecto para recibir eventos en tiempo real.
heartbeat	Señal periódica enviada por el cliente para mantener viva la conexión.
request-refresh	El cliente solicita al servidor que actualice o reenvíe información.
request-online-users	El cliente solicita la lista actual de usuarios conectados.

Fuente: Los autores.

Tabla 6.2: Evento Servidor → Cliente.

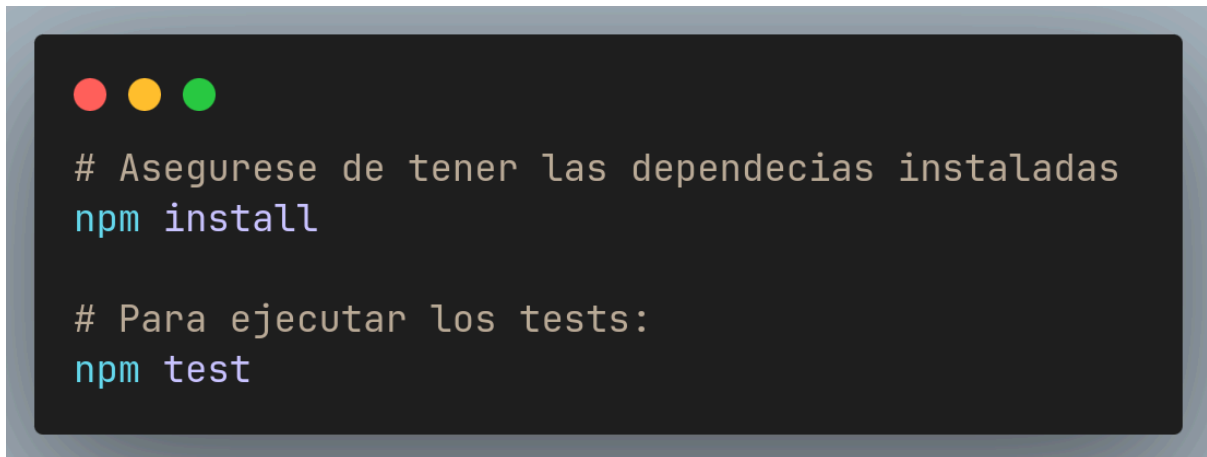
Evento	Descripción
joined-project	Confirmación de que el cliente se unió correctamente al proyecto.
user-joined	Notificación cuando otro usuario se une al mismo proyecto.
online-users	Envío de la lista actualizada de usuarios en línea.
task-created	Notificación en tiempo real de una nueva tarea creada.
task-updated	Notificación cuando una tarea ha sido actualizada.
task-deleted	Notificación de que una tarea ha sido eliminada.
comment-added	Notificación cuando se añade un nuevo comentario a una tarea.
refresh-tasks	Solicitud al cliente para que recargue las tareas (forzar sincronización).

Fuente: Los autores.

CAPÍTULO 7: EJECUCIÓN DE TEST Y SOLUCIÓN DE PROBLEMAS

7.1 Ejecución de test

Figura 7.1: Ejecutar test del sistema.

A terminal window with a dark background and light gray text. At the top left, there are three colored circles: red, yellow, and green. The text inside the terminal reads:

```
# Asegurese de tener las dependencias instaladas  
npm install  
  
# Para ejecutar los tests:  
npm test
```

Fuente: Los autores.

7.2 Solución de problemas comunes

1) Error de conexión a la base de datos

- Asegúrate de que SQL Server está en ejecución
- Verifica las credenciales en .env
- Confirma que el puerto coincide con el de tu servidor SQL
- Revisa que el usuario tenga permisos suficientes

2) Los correos no se envían

- Verifica la configuración SMTP en .env
- Si usas Gmail, usa una "Contraseña de aplicación"
- Consulta los logs del servidor

3) No se actualizan los datos en tiempo real

- Comprueba la conexión Socket.IO en la consola del navegador
- Revisa errores en la consola del servidor
- Asegura que el cliente esté unido al proyecto

CONCLUSIÓN

La incorporación de WebSockets mediante Socket.IO representa un componente clave dentro del diseño del sistema *task_manager*, ya que permite mejorar significativamente la experiencia de usuario al brindar interacciones en tiempo real. Esto no solo contribuye a una interfaz más dinámica, sino que también facilita la colaboración entre múltiples usuarios en un entorno sincronizado y coherente.

La arquitectura basada en eventos utilizada en Socket.IO promueve una separación clara de responsabilidades y mejora la escalabilidad del sistema. Cada evento está diseñado para cumplir una función específica dentro del ciclo de vida del proyecto o tarea, y su implementación modular permite mantener un código limpio, extensible y fácil de mantener.