Documento arquitectónico

Arquitectura de Software

Profesora: ALEXANDRA GUERRERO BOCANEGRA

Estudiantes: Dayana García

Gustavo Suarez Llano

Cristian Pereira Sánchez

Cristian Camilo Urbina Mera

Institución Universitaria Pascual Bravo Medellín 2025

Introducción al negocio

El presente documento describe el desarrollo de un sistema de gestión de tareas para la empresa Limpiar S.A., construido con el propósito de mejorar la organización del trabajo, facilitar el seguimiento de actividades y apoyar la coordinación entre los equipos. Para su diseño aplicamos los principios básicos de la arquitectura de software, con el fin de crear una solución estructurada, entendible y preparada para crecer según las necesidades de la empresa.

Durante el proceso analizamos el funcionamiento del negocio, los actores que intervienen y los procesos que deben ser soportados por la aplicación. A partir de esta comprensión definimos el modelo de dominio, identificando entidades como usuarios, tareas, estados y notificaciones, las cuales permiten representar de forma clara la información principal del sistema.

En la construcción del diseño incluimos diferentes patrones de software que ayudan a organizar mejor el código y mejorar la calidad del proyecto. Entre ellos se encuentran Factory Method, Repository, Observer, Strategy, Singleton, Command y Template Method, los cuales permiten manejar la creación de objetos, separar la lógica de acceso a datos, gestionar notificaciones, generar reportes y controlar acciones del sistema de una forma más ordenada y flexible.

Además, representamos la arquitectura mediante diagramas UML y el modelo C4 para mostrar cómo funciona el sistema desde lo general hasta sus partes internas. También seleccionamos el estilo arquitectónico adecuado de acuerdo con los requisitos funcionales y no funcionales, buscando siempre claridad, rendimiento y facilidad de mantenimiento.

En conjunto, este documento reúne el análisis, los modelos, las decisiones técnicas y la fundamentación del sistema, demostrando la aplicación práctica de lo aprendido en arquitectura de software y la capacidad para estructurar una solución completa y coherente para una necesidad real.

Descripción del contexto y problema a resolver

Limpiar S.A.S es una empresa dedicada a brindar servicios de aseo y mantenimiento en oficinas. Actualmente, la empresa enfrenta dificultades en la organización de las actividades diarias de sus empleados, ya que gran parte de la planificación se hace de manera manual.

Este manejo poco centralizado genera problemas como:

• Pérdida de información sobre tareas asignadas.

• Dificultades para hacer seguimiento al cumplimiento de las labores.

• Sobrecarga de trabajo en algunos empleados por falta de control en la distribución de actividades.

Para dar solución a estos inconvenientes, se plantea el desarrollo de una aplicación de gestión de tareas, que permita mejorar la coordinación, visibilidad y control de las actividades dentro de la empresa.

1.Identificación de objetivos estratégicos

El proyecto busca alcanzar los siguientes objetivos estratégicos:

1. Optimizar la organización interna: centralizar la asignación y seguimiento de tareas en una sola plataforma.

2. Mejorar la eficiencia operativa: reducir el tiempo perdido en coordinación manual y minimizar errores humanos.

3. Garantizar el cumplimiento de actividades: establecer recordatorios y notificaciones para que los empleados ejecuten sus labores a tiempo.

4. Fortalecer la comunicación interna: permitir que supervisores y empleados tengan claridad sobre las responsabilidades asignadas.

5. Incrementar la satisfacción del cliente: asegurar que los servicios de aseo se realicen con puntualidad y calidad, reflejando una mejor imagen de la empresa.

Análisis de actores y procesos clave Clasificación de actores

Actores primarios (usan directamente el sistema):

- Supervisor de Operaciones: asigna tareas, supervisa avances y valida cumplimiento.

- Empleado de Aseo: recibe las tareas asignadas, las ejecuta y reporta su finalización.

Actores secundarios (interactúan in-directamente con el sistema):

Cliente (empresas, instituciones): recibe los servicios de aseo y puede dar retroalimentación sobre la calidad del trabajo.

Actores de soporte (aseguran el correcto funcionamiento del sistema):

Administrador del Sistema/TI: mantiene la aplicación funcionando, gestiona usuarios y corrige errores técnicos.

Procesos críticos que el sistema debe soportar

1. Gestión de usuarios: registro, edición y control de accesos para empleados y supervisores.

2. Asignación de tareas: el supervisor crea y asigna tareas específicas a los empleados.

3. Seguimiento de tareas: actualización de estado (pendiente, en proceso, completada).

4. Estatus de tarea asignada: informes sobre cumplimiento de tareas, tiempos de respuesta y desempeño de empleados.

5. Calificación a través de encuesta: registro de comentarios sobre la calidad del servicio.

Matriz de actores y procesos (diagrama simplificado)

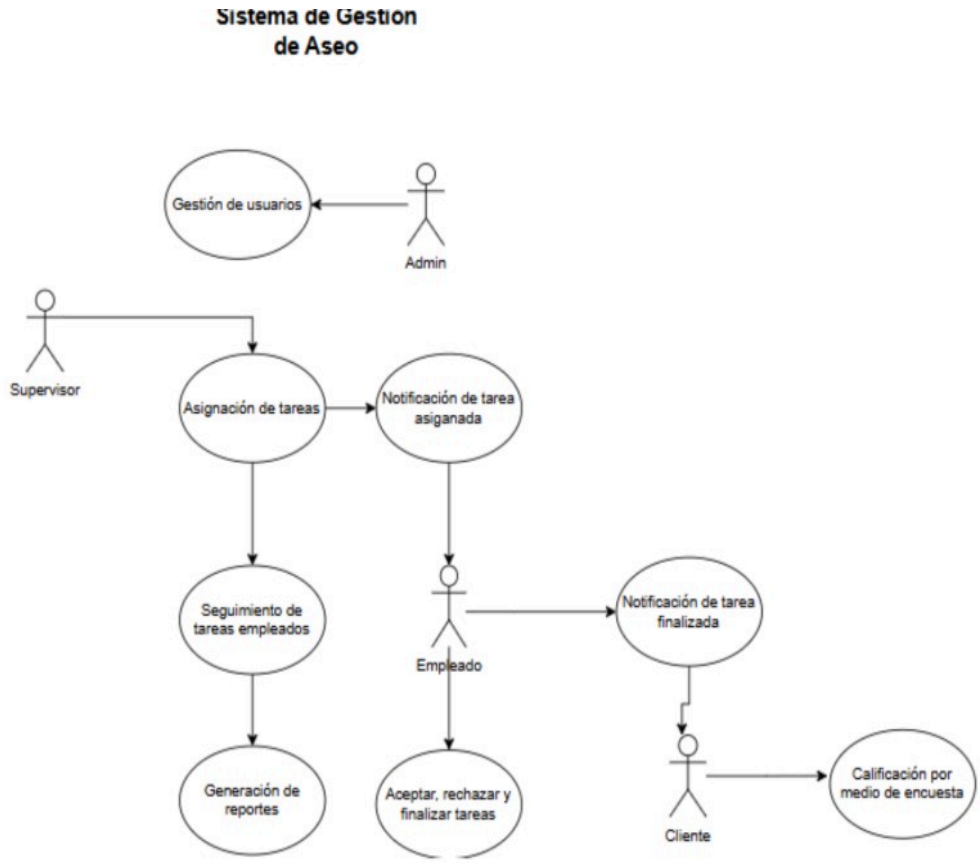


Ilustración 1 Diagrama simplificado

3. Modelo de dominio inicial

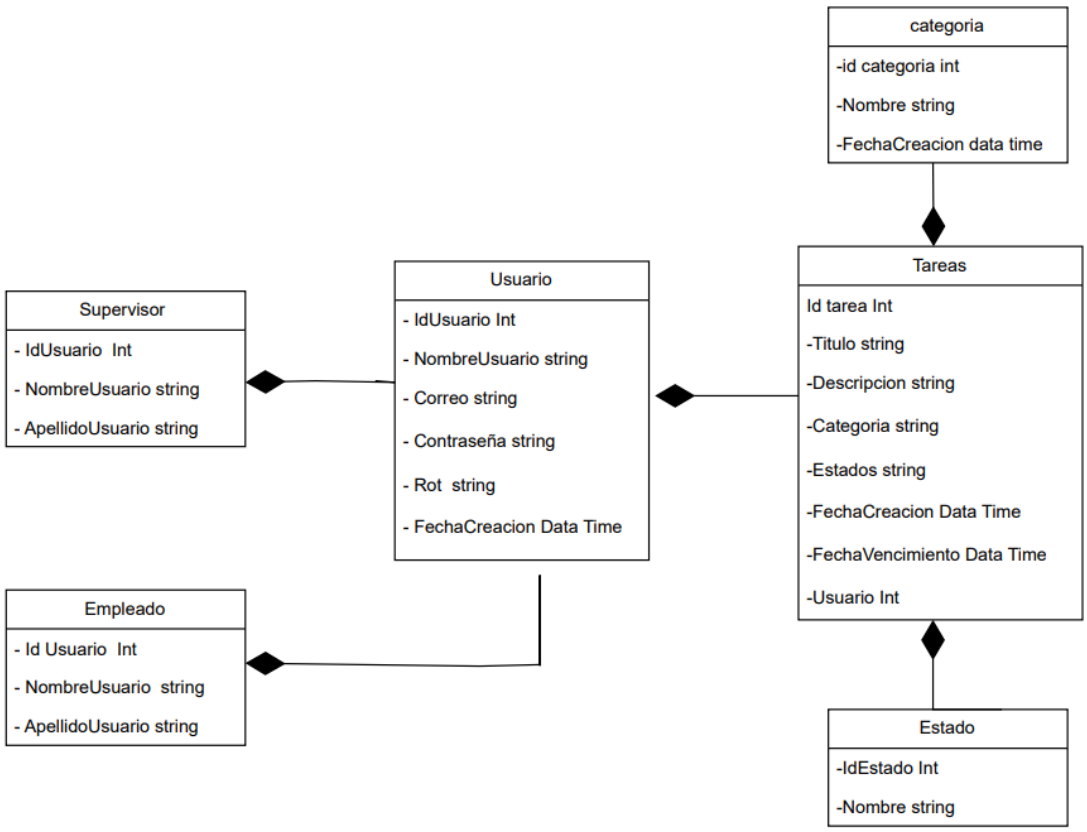


Ilustración 2 Modelo de dominio inicial

Entidades principales y atributos básicos

1. Usuario

- o IdUsuario
- o NombreUsuario
- o Contraseña
- o Correo
- o Rol
- o FechaCreación

2. Tarea

- o IdTarea
- o Título
- o Descripción
- o CategoriaId
- o UsuarioId

- o EstadoId
- o FechaCreacion
- o FechaVencimiento

3. Categoría

- o IdCategoria
- o Nombre
- o Fecha de creación

4. Estado (Pendiente, En proceso, Completada)

- o IdEstado
- o Nombre

Relaciones y multiplicidades

• Usuario (Supervisor) asigna muchas Tareas → (1 Supervisor: * Tareas).

• Usuario (Empleado) ejecuta muchas Tareas → (1 Empleado: * Tareas).

• Una Tarea puede tener varios estados → (1 Tarea: * Reportes).

• Una Tarea puede tener varias Categoría → (1 Categoría: * Tareas).

Diagrama de clases UML (versión conceptual)

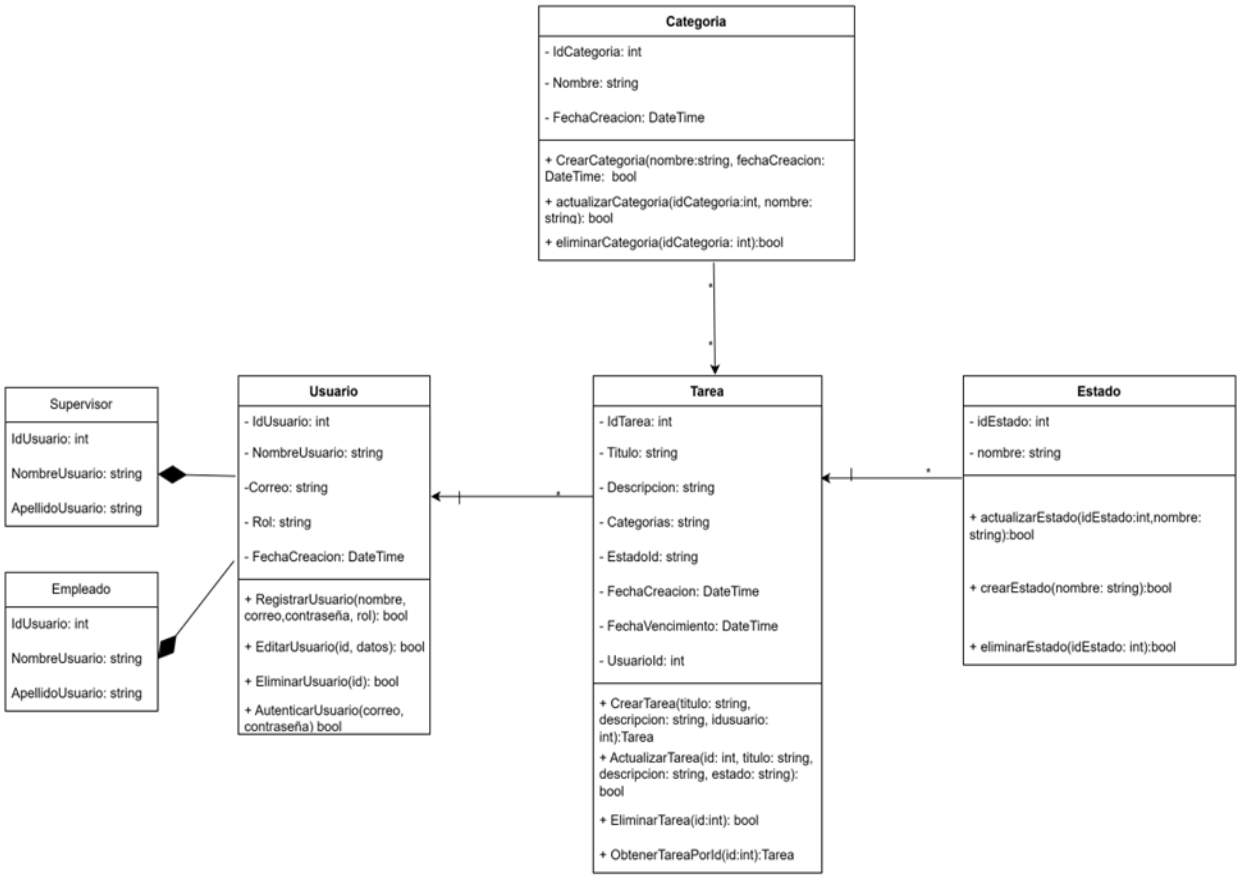


Ilustración 3 Diagrama de clases UML

5. Validación del modelo

Contraste con requisitos funcionales

El modelo planteado se revisa frente a los principales requisitos funcionales:

• **Gestión de usuarios:** el modelo contempla la entidad Usuario con roles (Supervisor, Empleado, Administrador), cumpliendo con la necesidad de control de acceso.

• **Asignación de tareas:** se cubre con las entidades Tarea.

• **Seguimiento de tareas:** los estados de la entidad Tarea (Pendiente, En proceso, Completada) permiten un control del avance.

Relación con requisitos no funcionales

• Seguridad:

o Se garantiza mediante la gestión de roles en la entidad Usuario, evitando accesos indebidos.

o Se realiza consumo de API mediante Token en Swagger.

• Rendimiento:

o El modelo es sencillo, con relaciones claras que permiten consultas

eficientes. o Se puede optimizar con índices en atributos clave

(IdUsuario, IdTarea).

• Escalabilidad:

o El sistema puede crecer para soportar múltiples sedes

o clientes

• Disponibilidad:

o El modelo soporta replicación de datos y puede integrarse con

servicios en la nube para asegurar acceso 24/7.

Ajustes propuestos con base en feedback

Durante la validación, se identifican posibles mejoras:

1. **Notificaciones como entidad independiente:**

o Crear una entidad Notificación con atributos (IdNotificación, Mensaje, Fecha, EstadoLeído, IdUsuario) para mejorar el control de recordatorios.

2. Historial de cambios en tareas:

o Registrar modificaciones en estado y asignaciones para tener trazabilidad de cada tarea.

3. **Roles configurables:**

o Permitir que los roles no estén fijos en código, sino parametrizados en la base de datos, facilitando cambios futuros.

5. Documento de atributos priorizados

Selección de atributos de calidad relevantes

Para este proyecto se identifican los siguientes atributos de calidad como los más importantes:

• Disponibilidad

• Escalabilidad

• Seguridad

• Rendimiento

Priorización con justificación desde el negocio

1. **Disponibilidad (Alta prioridad)**

o Justificación: los empleados de aseo y supervisores deben acceder al sistema en todo momento para consultar y actualizar tareas. Si el sistema está caído, afecta directamente la operación diaria y la satisfacción del cliente.

2. **Seguridad (Alta prioridad)**

o Justificación: el sistema gestiona información sensible (datos de clientes, reportes de empleados y credenciales de acceso). Una vulnerabilidad podría comprometer la confianza y reputación de la empresa.

3. **Rendimiento (Media-Alta prioridad)**

o Justificación: el sistema debe responder de manera ágil al asignar tareas. Si el tiempo de respuesta es lento, se reduce la productividad de los empleados y supervisores.

4. **Escalabilidad (Media prioridad)**

o Justificación: actualmente la empresa atiende un número limitado de clientes, pero a futuro puede expandirse a nuevas ciudades o aumentar la cantidad de empleados. El sistema debe poder crecer sin requerir rediseños completos. Definición de métricas asociadas

Definición de métricas asociadas

• Disponibilidad:

99.5% mensual (máximo 3.6 horas de inactividad al mes).

• Seguridad:

o Autenticación mediante credenciales seguras (mínimo 8 caracteres, combinación de letras y números).

o Autorización basada en roles (un empleado no puede ver información de otro equipo).

o Encriptación de datos sensibles (contraseñas hash, comunicación

HTTPS).

• Rendimiento:

tiempo de respuesta promedio de < 2 segundos en consultas y

asignaciones.

• Escalabilidad:

o Soportar hasta 500 usuarios concurrentes en la primera

versión.

o Escalar hasta 2000 usuarios con mejoras de infraestructura en

la nube.

Relación clara entre drivers de negocio y métricas de

calidad

Driver de negocio	Atributo de calidad	Métrica
Garantizar continuidad del servicio de aseo	Disponibilidad	≥ 99.5% de uptime mensual
Proteger información de empleados y clientes	Seguridad	Autenticación robusta, autorización por roles, encriptación
Asegurar eficiencia en la operación	Rendimiento	Tiempo de respuesta < 2 segundos en operaciones críticas
Posibilidad de crecimiento empresarial	Escalabilidad	Hasta 2000 usuarios concurrentes en futuras fases

Incorporación de Patrones de Software al Diseño

Selección de patrones de diseño

Entidades: Usuarios y Tareas

Patrones de diseño: Factory Method y Repository.

Patrón Factory Method (Creacional)

La razón de usar este patrón es que permite centralizar la creación de objetos y dar más flexibilidad al sistema. Con el Factory Method no se crean las instancias de manera directa en cualquier parte del código, sino que se definen fábricas encargadas de esa responsabilidad. Esto resulta útil porque deja el diseño preparado para cambios futuros, como la posibilidad de manejar diferentes tipos de usuarios o tareas con características adicionales, sin necesidad de modificar la lógica en múltiples lugares.

Patrón Repository (Estructural)

El propósito de este patrón es separar la lógica de acceso a datos de la lógica de negocio, de modo que las clases que gestionan Usuarios y Tareas no dependan directamente del DbContext. En lugar de acceder de forma directa a la base de datos, se trabaja a través de repositorios que actúan como una capa intermedia.

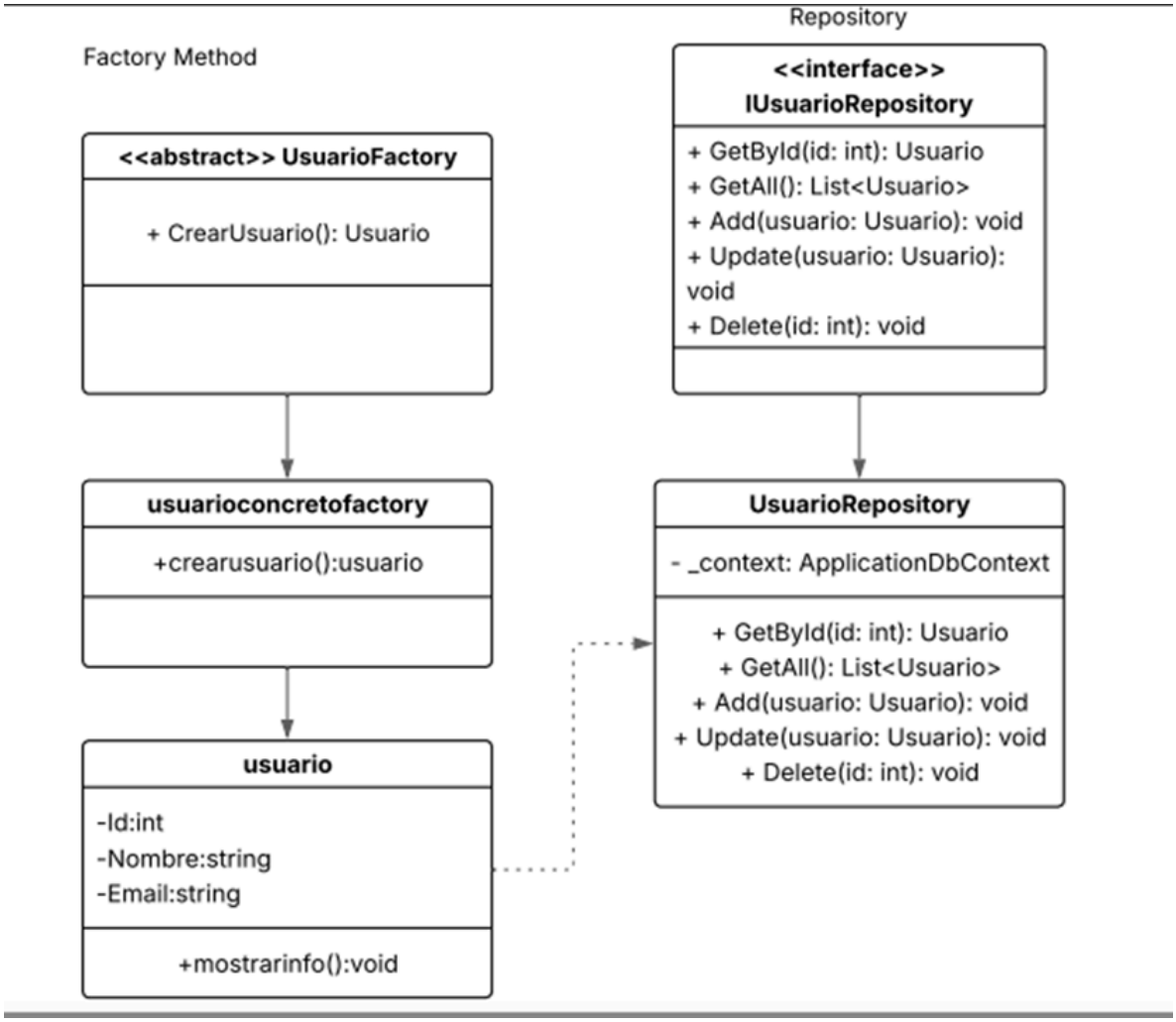


Ilustración 4 Patrones de diseño

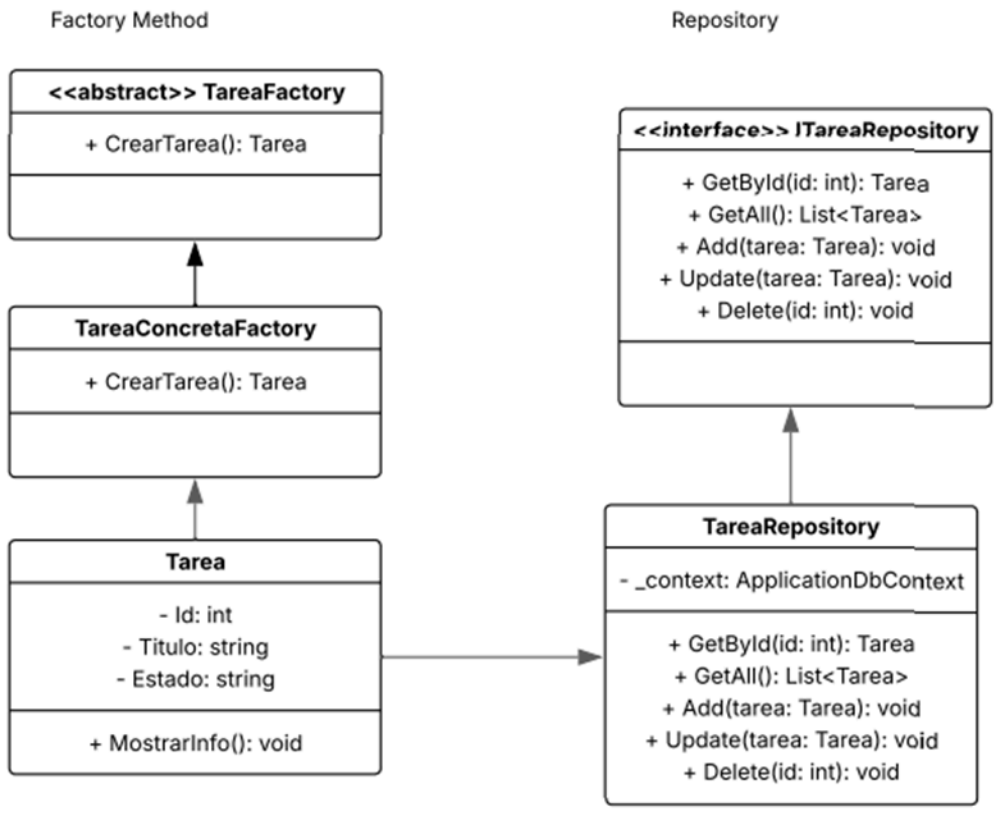


Ilustración 5 Patrones de Diseño

Módulo supervisor

Singleton y command

Patrón Singleton (Creacional)

Este patrón se usa porque nos asegura un TaskManager, es palabras sencillas es quien puede guardar y gestionar todas las tareas, evitando duplicidad o inconsistencias en la información.

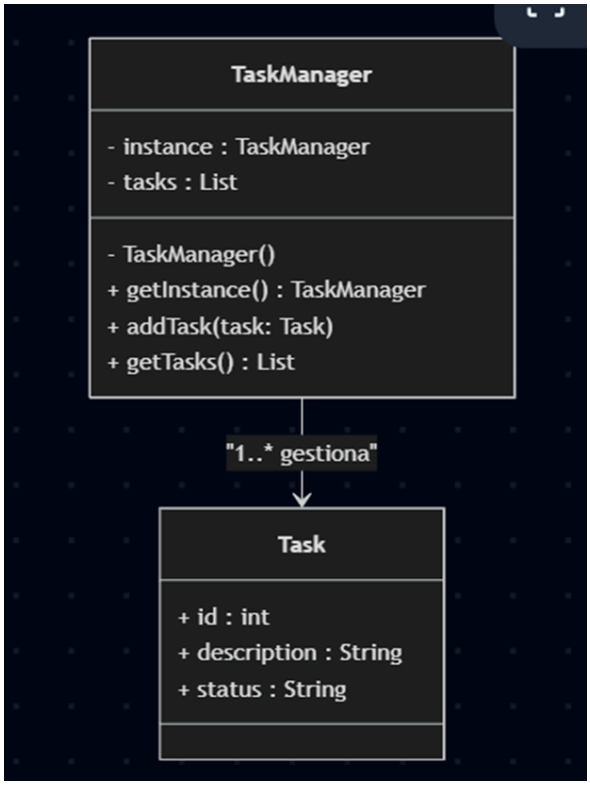


Ilustración 6 Diagrama UML de la relación entre TaskManager y Task

Patrón Command (patrón de comportamiento)

Se usa este patrón para separar el "qué hacer" del "quién lo hace", ya que a través de de este patrón podemos implementar guardar historial que es útil para auditorias, implementación del deshacer y rehacer, en este caso el command nos ayuda a encapsular acciones del supervisor, como asignar tarea, visualizar tarea y filtrar tareas según su estado.

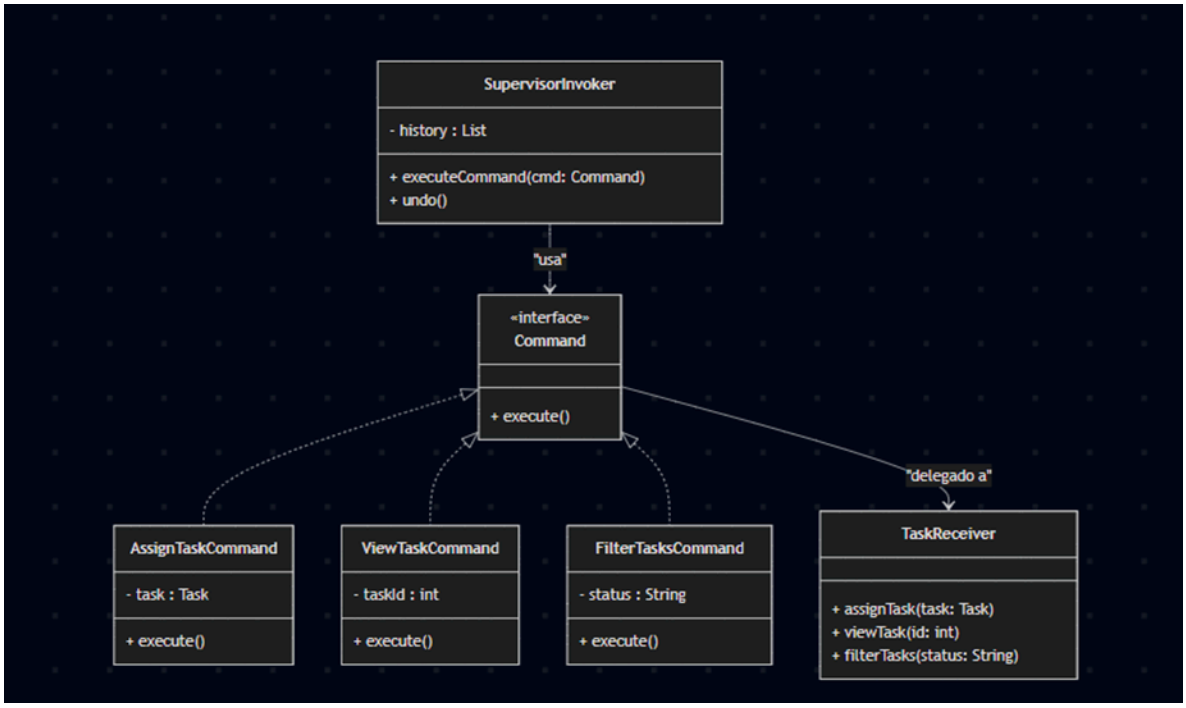


Ilustración 7 Diagrama UML del patrón Command aplicado a la gestión de tarea

Notificaciones

Observer

●El patrón observer se ajusta a la funcionalidad de notificaciones, ya que se utiliza principalmente cuando hay una necesidad de notificar cambios de estado de algún objeto o acción de una funcionalidad.

●Por lo tanto encaja perfecto en nuestra funcionalidad para que el supervisor pueda recibir las alertas de los cambios de estado de las tareas, y también poder avisar al empleado cuando se le ha asignado una tarea.

●El patrón Publish-Subscribe muy similar al observer en su principio de comunicación entre objetos para notificar cambios, pero difieren en que el observer los usuarios se deben registrar directamente al sujeto, y el publish-subscribe utiliza un broker (tercero) para gestionar las suscripciones para la distribución de los mensajes o alertas, por eso decidimos utilizar observe.

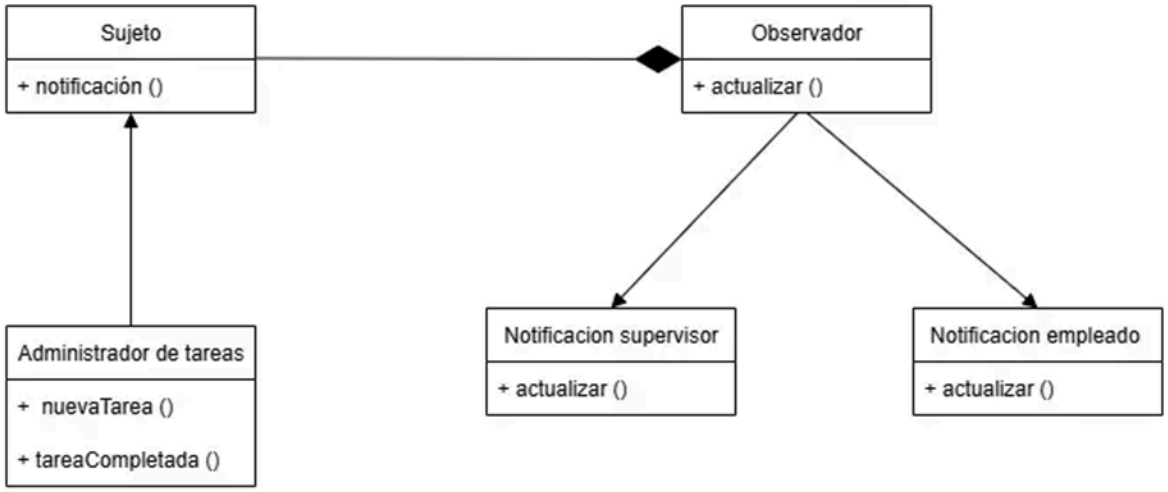


Ilustración 8 Diagrama de clases del Patrón Observador (Observer)

El administrador de tareas es el sujeto cuando pasa algo (nueva tarea, tarea completada), ejecuta notificación.

Notificación al supervisor y notificación al empleado son observadores. reciben la notificación y muestran o envían la notificación.

Generar reportes

Strategy

- El perfil de supervisor puede necesitar exportar reportes en diferentes formatos.

- El patrón Strategy por su funcionalidad permite tomar algo específico en este caso la funcionalidad de generar reportes y cada formato colocarlo en clases separadas que se denominan estrategias, es más dinámico y así generar el reporte con un formato específico según necesidad que pueden ser (Excel, PDF)

- El patrón de diseño Builder es otra opción para esta funcionalidad pero en esté se deben construir los objetos paso a paso, el patrón de diseño Builder es más Creacional mientras que el Strategy es más comportamental por lo que nos parece más apto para la aplicación.

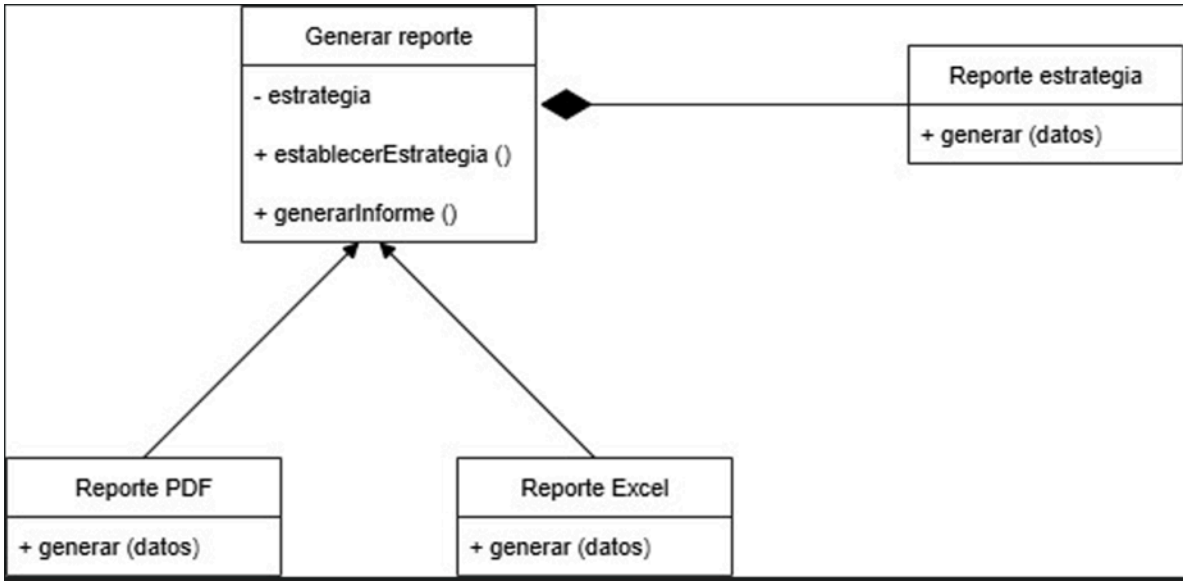


Ilustración 9 Diagrama de clases del Patrón Estrategia

Calificación mediante encuestas de satisfacción

Patrón de diseño: Template Method

Se requiere un sistema que estandarice las encuestas de satisfacción, manteniendo un flujo común pero permitiendo adaptar preguntas y cálculos según el servicio evaluado.

Por que: El método de plantilla es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase, pero permite que las subclases anulen pasos específicos del algoritmo sin cambiar su estructura.

Una encuesta tiene una estructura fija (mostrar preguntas, capturar respuestas, calcular promedio, mostrar resultado). Pero el contenido de la encuesta (preguntas, ponderación, cálculo) puede variar según el tipo de encuesta (ejemplo: satisfacción de servicio, limpieza, puntualidad).

Ilustración 10 Diagrama de clases del Patrón Template Method

Survey Template define el flujo general:

1. Mostrar preguntas
2. Recoger respuestas
3. Calcular puntaje
4. Mostrar resultado

Cleaning Survey y PunctualitySurvey definen las preguntas y la forma de calcular la calificación.**Justificación**

Encuestas (Template Method):

Permite codificar el flujo común una sola vez y obliga a las subclases a implementar solo los pasos que varían.

Todas las encuestas siguen un flujo común (mostrar preguntas, recolectar respuestas,

calcular resultado), pero cada tipo de encuesta puede personalizar pasos específicos sin alterar la estructura general.

Modelado Arquitectónico con UML y Modelo C4

arquitectura del sistema mediante el uso integrado del Modelo C4 y los diagramas UML, con el fin de representar la solución desde diferentes niveles de abstracción y asegurar coherencia entre la vista conceptual, lógica y física del proyecto.

Fase 1 – Modelado con el Modelo C4 (Vista conceptual y lógica)

1. Nivel 1 – Contexto: representar el sistema dentro de su entorno, actores externos y sus interacciones.

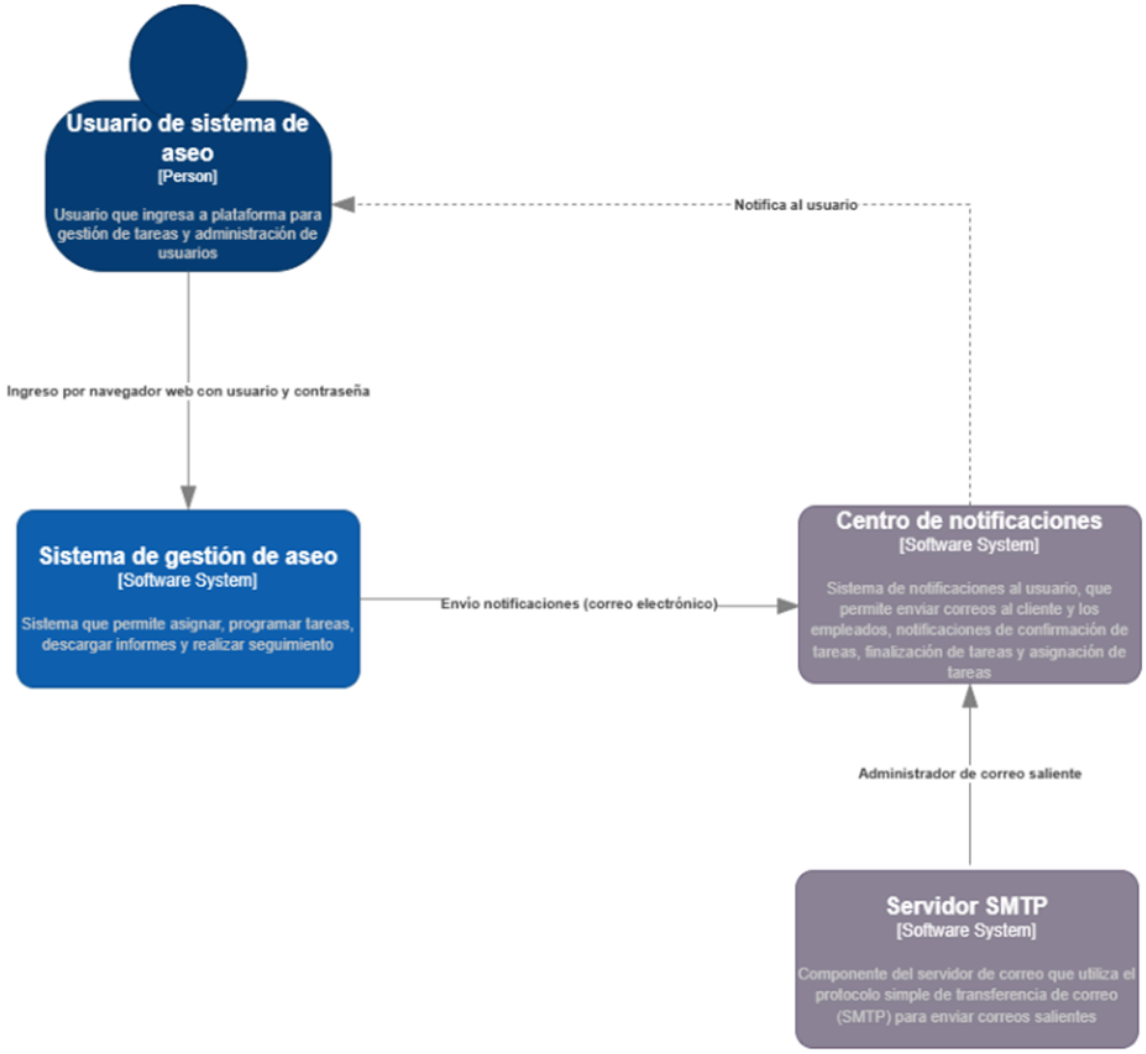


Ilustración 11 Modelado con el modelo C4 nivel 1

Ilustración 12 Modelado con el Modelo C4 (Vista conceptual y lógica)

2. Nivel 2 – Contenedores: describir las principales aplicaciones, servicios, bases de datos o API que conforman el sistema.

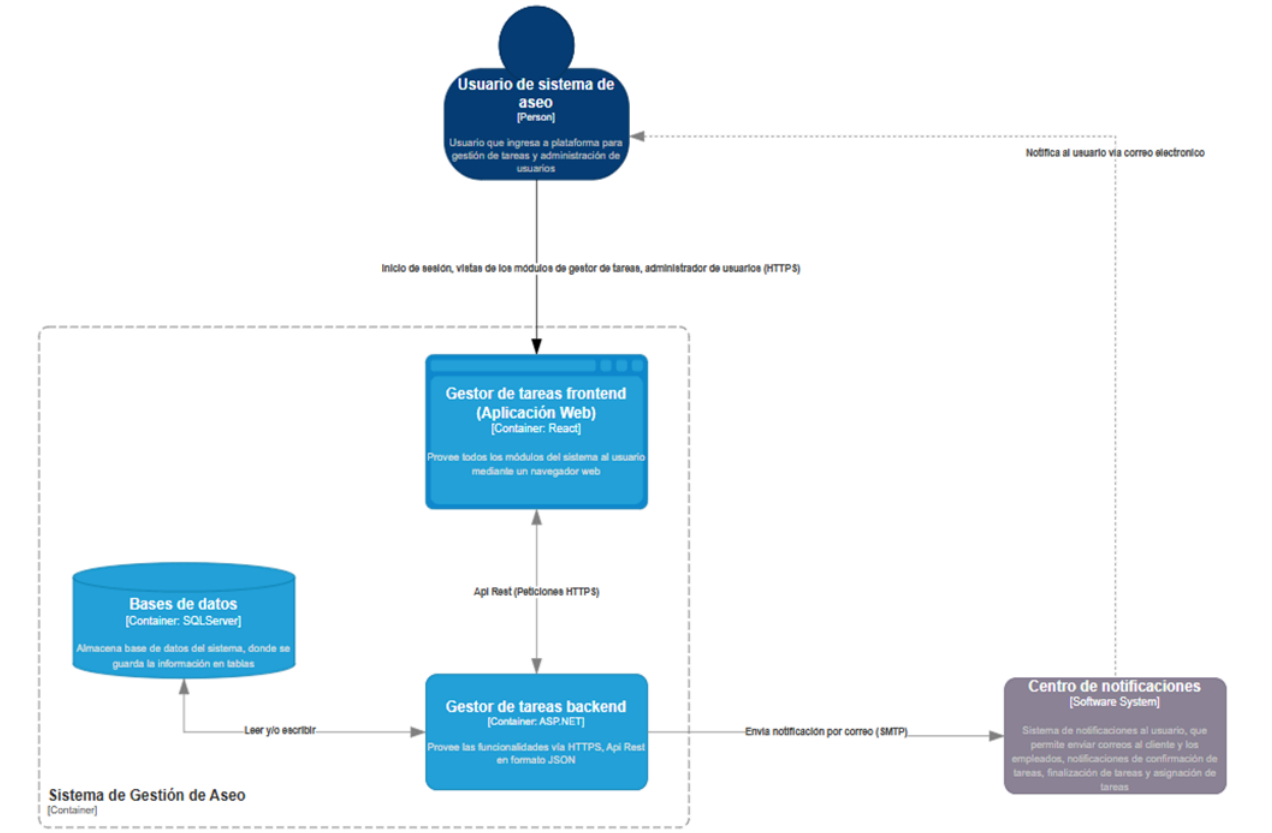


Ilustración 13 Modelado con el modelo C4 nivel 2

3. Nivel 3 – Componentes: detallar la estructura interna de un contenedor clave (por ejemplo, backend o servicio principal), identificando módulos, repositorios, servicios y controladores.

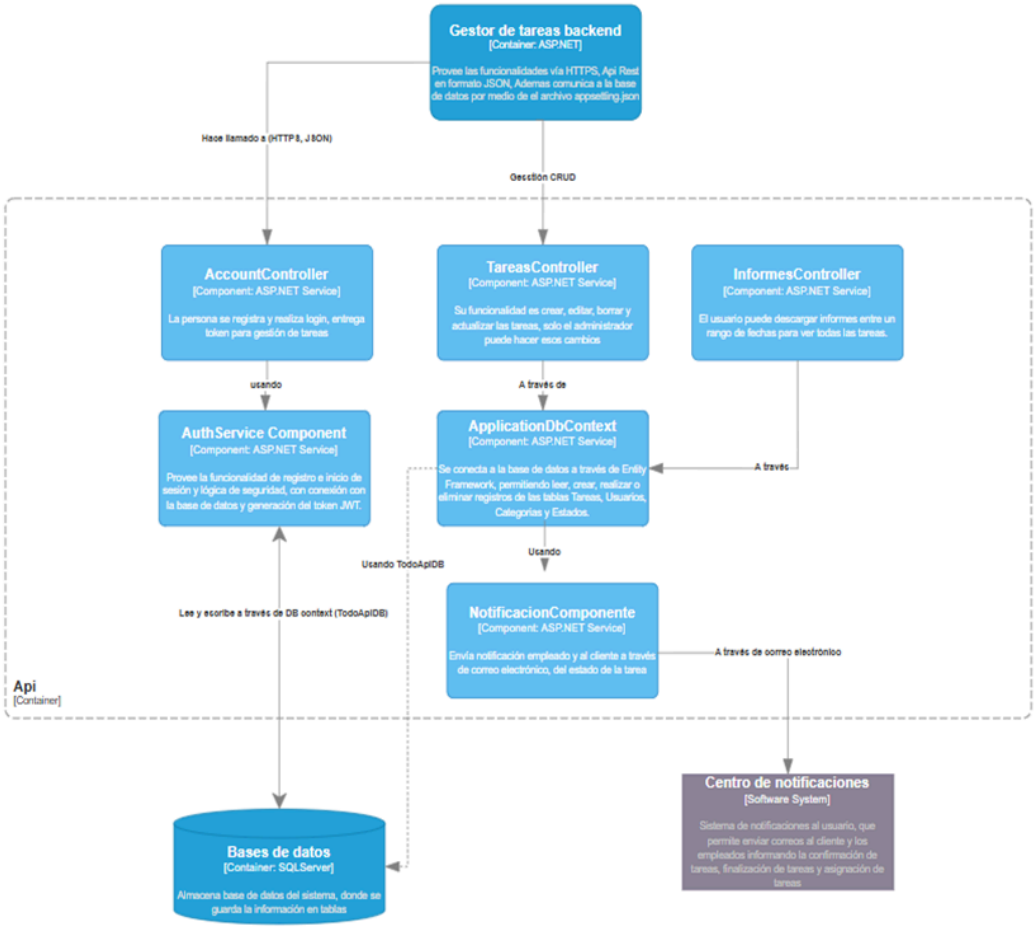


Ilustración 14 Modelado con el modelo C4 nivel 3

Fase 2 – Diagramas UML (Vista de diseño y despliegue)

1. Diagrama de Componentes: mostrar la estructura lógica y dependencias entre módulos o subsistemas.

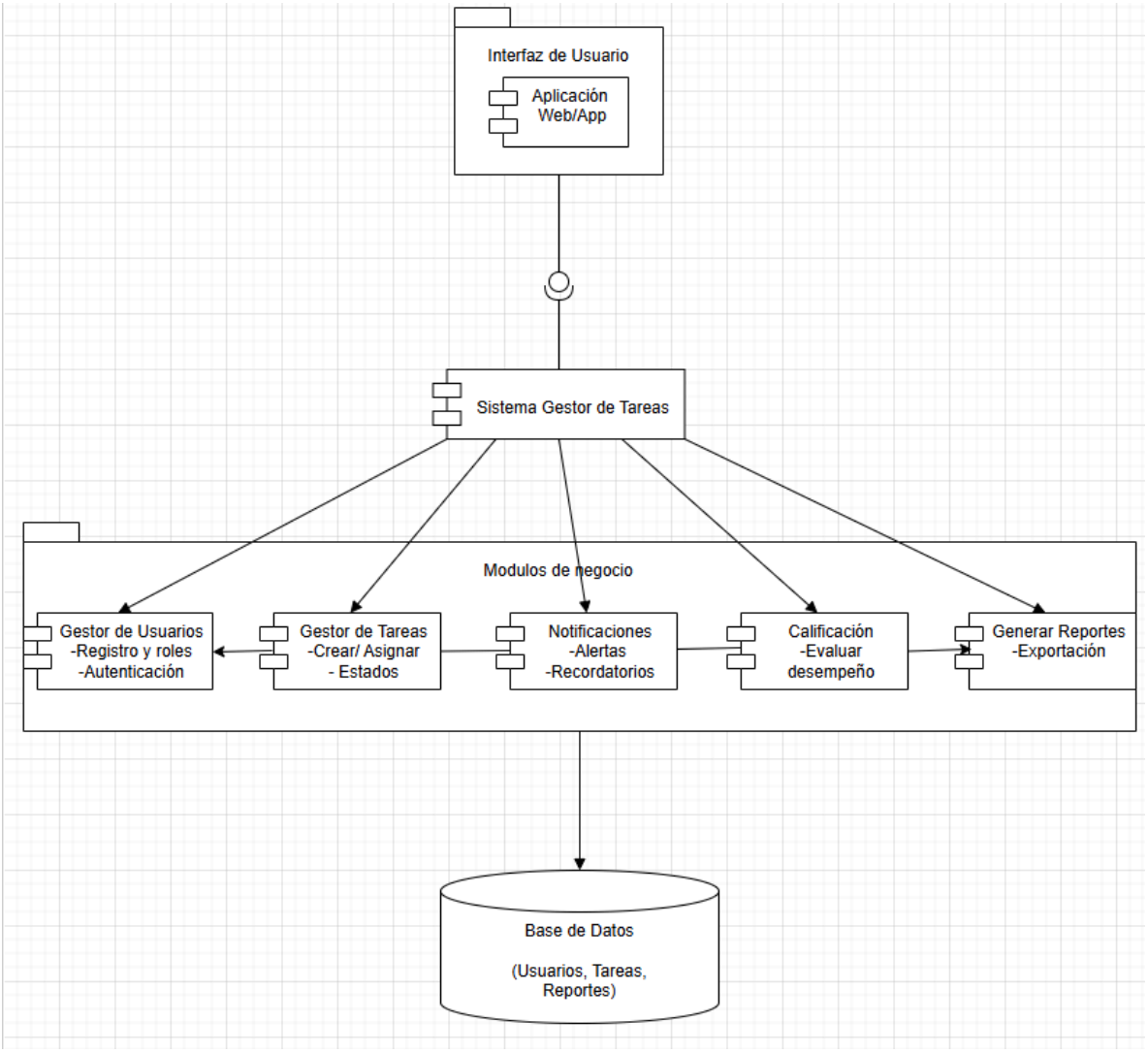


Ilustración 15 Diagrama de Arquitectura de Módulos del Sistema Gestor de Tareas

2. Diagrama de Clases: representa la capa de negocio aplicando los patrones de diseño definidos en el entregable anterior, con clases, interfaces y relaciones correctamente modeladas.

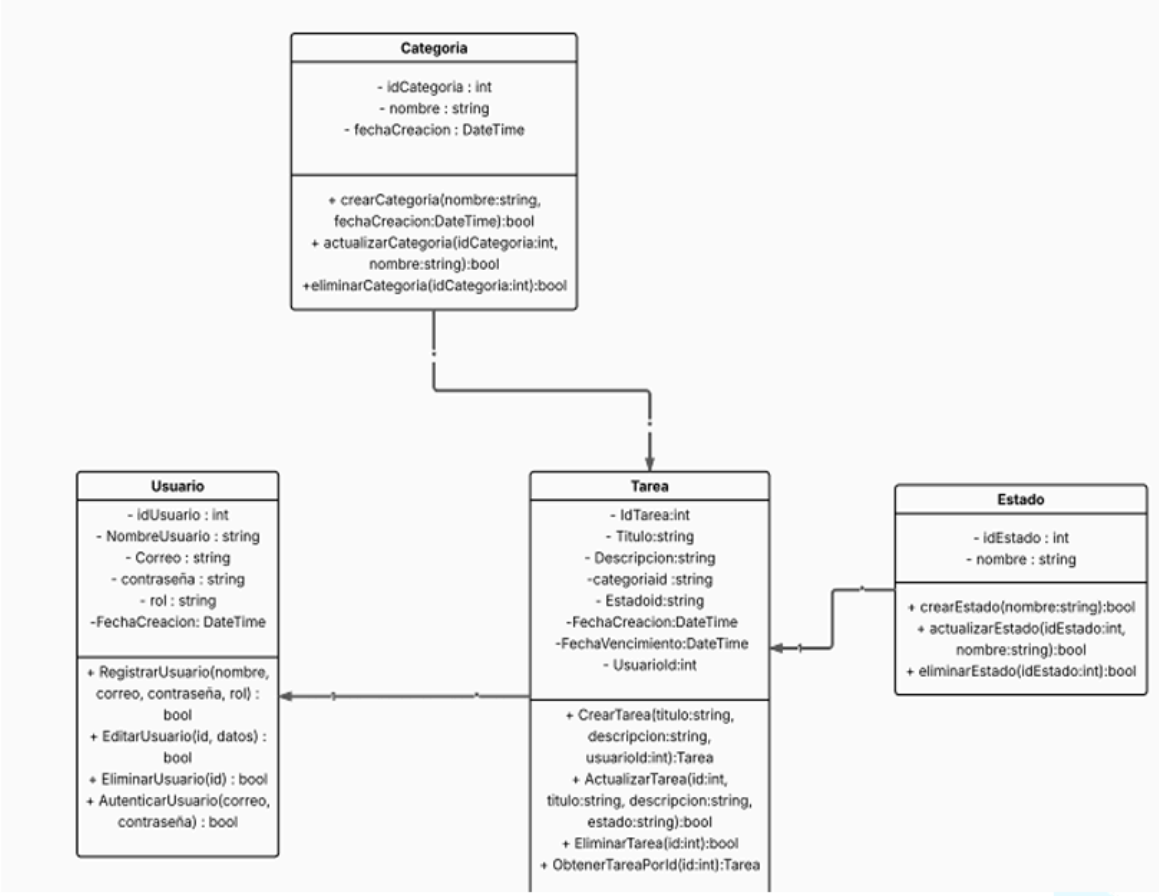


Ilustración 16 Diagrama de clases

3. Diagrama de Despliegue: representar la infraestructura física y virtual (nodos, servidores, base de datos, nube, clientes, conexiones).

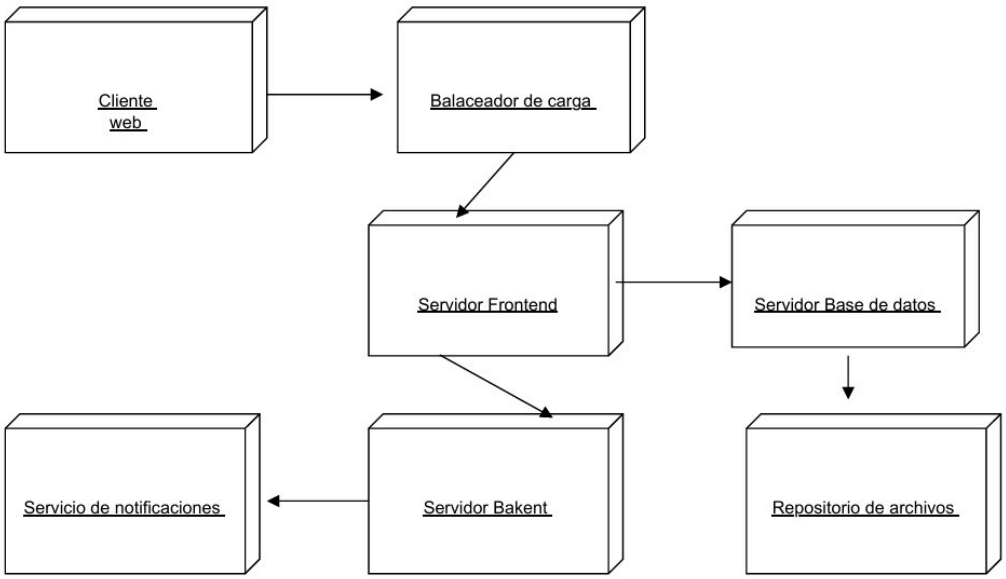


Ilustración 17 Diagrama de despliegue

Fase 3 – Análisis y justificación técnica

Redactar una explicación técnica (máx. una página) que contenga:

• Justificación de los patrones aplicados en el diagrama de clases.

• Coherencia entre los niveles del modelo C4 y los diagramas UML.

• Principios de diseño y decisiones arquitectónicas (modularidad, escalabilidad, separación de capas).

Análisis y justificación técnica

La arquitectura del sistema de gestión de tareas para la empresa Limpiar S.A. se diseñó aplicando principios y patrones que permiten crear una solución ordenada, clara y fácil de mantener. En el diagrama de clases se utilizaron varios patrones de diseño para resolver necesidades específicas del sistema. El patrón Factory Method ayuda a crear objetos como tareas o usuarios sin depender directamente de sus constructores, lo que facilita extender el sistema cuando aparezcan nuevos tipos de tareas. El patrón Repository organiza el acceso a datos y separa la lógica de negocio de la base de datos, permitiendo que el backend se mantenga modular y que los cambios en SQL Server no afecten el código principal. El patrón Observer permite manejar el envío de notificaciones de una forma flexible, de modo que cada vez que una tarea cambia de estado, los observadores pueden recibir alertas por correo o por otros medios. También se usó Strategy para la generación de reportes y Template Method para procesos que requieren pasos similares, como validaciones o flujos repetitivos dentro del sistema.

La relación entre los niveles del modelo C4 y los diagramas UML es coherente porque cada vista representa el sistema con un nivel de detalle distinto, pero sin perder consistencia. En el nivel de contexto (C4) se muestra cómo el usuario interactúa con el frontend y cómo este se comunica con el backend. En el nivel de contenedores se identifican los elementos principales: cliente, servidor frontend, servidor backend y la base de datos. Finalmente, el nivel de componentes detalla servicios, controladores, repositorios y manejadores internos del backend. Estos componentes aparecen representados en los diagramas UML de clases y componentes, lo que muestra que ambos modelos describen las mismas partes del sistema, solo que desde diferentes perspectivas.

Las decisiones arquitectónicas se tomaron buscando modularidad, separación de responsabilidades y escalabilidad. El sistema se organizó por capas (presentación, lógica de negocio y datos), lo que facilita el mantenimiento y permite trabajar en cada parte sin afectar a las demás. Esta separación también hace que el sistema sea más fácil de probar y de extender. Además, se seleccionó una arquitectura cliente–servidor que permite distribuir la carga y manejar el frontend y el backend como elementos independientes. Cada módulo del backend se diseñó para cumplir una sola responsabilidad, aplicando principios como SRP y Open/Closed, lo que permite agregar nuevas funciones sin modificar las ya

existentes. En conjunto, estas decisiones garantizan que el sistema pueda crecer, cambiar o integrarse con otros servicios sin perder estabilidad ni claridad en su estructura.

Selección del Estilo Arquitectónico principal

Analizar los requerimientos del sistema propuesto (funcionales y no funcionales).

● Funcionales

- Generar reportes

La generación de reportes requiere procesar información desde la base de datos y aplicar distintas lógicas según el tipo de reporte.Esto influye en la arquitectura porque es necesario tener una lógica centralizada en el servidor que procese los datos y evite que el cliente haga cálculos pesados. Esto se ajusta bien al modelo Cliente–Servidor, donde el backend es el encargado de procesar y enviar los resultados al cliente de forma segura.

- Realizar CRUD a las tareas

Las operaciones CRUD (crear, leer, actualizar, eliminar) necesitan un servidor que gestione las solicitudes, aplique validaciones y mantenga integridad en la base de datos.
Por esta razón, se requiere una arquitectura donde el backend controle estas operaciones, lo que encaja directamente con Cliente–Servidor, ya que el servidor maneja toda la lógica y el cliente solo envía las peticiones.

- Autenticación de usuario

El inicio de sesión y la validación de credenciales deben realizarse en un entorno seguro para proteger los datos de los usuarios. Esto influye en la decisión arquitectónica porque se necesita un servidor que administre sesiones, tokens y reglas de seguridad. Por eso, el modelo Cliente–Servidor es adecuado, ya que el backend controla la autenticación y el frontend solo muestra la interfaz.

- Notificaciones

El sistema envía notificaciones cuando una tarea cambia de estado o se asigna. Esto obliga a que exista un componente central que detecte estos cambios y gestione el envío de

mensajes. En consecuencia, se necesita un servidor capaz de manejar eventos y ejecutar procesos automáticos, lo cual es natural dentro de la arquitectura Cliente–Servidor.

● No funcionales

- Seguridad

El sistema debe proteger información sensible como usuarios, contraseñas y tareas.
La arquitectura Cliente–Servidor permite manejar la seguridad desde el backend usando: cifrado, tokens, roles, validaciones. De esta forma, los datos nunca quedan expuestos en el cliente.

- Escalable

El sistema puede crecer en usuarios, tareas o módulos.
La arquitectura Cliente–Servidor facilita esta necesidad porque se puede:

- aumentar la capacidad del servidor
- agregar balanceadores de carga
- separar frontend y backend cuando sea necesario
  Por eso es una decisión adecuada para este proyecto.

- Usabilidad

El cliente solo debe preocuparse por mostrar una interfaz clara y fácil de usar. Esto se logra mejor cuando la lógica pesada vive en el servidor, y el cliente solo presenta resultados, lo cual es propio del modelo Cliente–Servidor.

- Mantenibilidad

Tener la lógica de negocio centralizada en el servidor facilita corregir errores o agregar funciones nuevas. En Cliente–Servidor solo se modifica el backend y todos los usuarios reciben el cambio automáticamente, sin necesidad de actualizar el cliente manualmente.

- Rápido

Un servidor dedicado permite procesar datos más rápido que un cliente distribuido.
Además, enviar solo la información necesaria al cliente reduce la carga y mejora los

tiempos de respuesta. Esto hace que la arquitectura Cliente–Servidor sea la opción más eficiente.

Revisa los estilos arquitectónicos vistos en clase:

• Cliente-Servidor
• Por Capas
• Orientado a Servicios (SOA)
• Microservicios
• Orientado a Eventos

Luego de comparar los estilos arquitectónicos, nos inclinamos por el estilo Cliente-Servidor, es el más adecuado para el sistema gestor de tareas, debido a su simplicidad, bajo costo, facilidad de mantenimiento.

Otros estilos como Microservicios presentan beneficios altamente escalables, pero son de una complejidad innecesaria para los requerimientos de este proyecto.

Selección al estilo que mejor se adapta al contexto del proyecto.

Cliente-Servidor

Justifica la elección

• Compatibilidad con las necesidades del sistema.

Se elige el estilo arquitectónico cliente - servidor ya que este modelo se aplica mejor por lo que requiere una clara separación entre la interfaz de usuario y los servicios de negocio. El cliente se encarga de la interacción con el usuario (visualización, creación y edición de tareas), mientras que el servidor administra los procesos, valida la información y gestiona el acceso a la base de datos de manera centralizada.

Adicionalmente este modelo facilita la mantenibilidad y la escalabilidad, ya que las actualizaciones del sistema pueden realizarse directamente en el servidor, sin afectar a los clientes. Además, ofrece mayor seguridad al controlar desde un punto central la autenticación de usuarios y los permisos sobre las tareas.

• Ventajas y limitaciones del estilo seleccionado.

Este estilo se caracteriza por dividir el sistema en 2 componentes principales: el cliente, que gestiona la interfaz y las interacciones con el usuario y el servidor, que se encarga de

procesar la lógica del negocio, almacenar los datos y responder a las solicitudes, es bastante utilizado en sistemas web, aplicaciones móviles y plataformas empresariales por su equilibrio entre simplicidad, control y eficiencia.

También se resalta la ventaja de separación de responsabilidades en cuanto a la distinción entre la capa de presentación y la lógica del negocio; permitiendo así la facilidad de los desarrolladores en cuanto al trabajo paralelo y acelera el mantenimiento del sistema, ya que los cambios de la interfaz no afectan la lógica interna ni la base de datos.

En cuanto a la centralización y control de datos, todo está concentrado en el servidor, lo cuál garantiza consistencia, seguridad y respaldo. Debido a este gran volumen de similaridades se elige este estilo y es acorde al gestor de tareas por lo que la interacción es entre el usuario y el cliente no de una manera directa, ya que la aplicación solo con la función de crear, editar o eliminar tareas y toda la información se actualiza en tiempo real.

• Limitaciones

Dependencia del servidor: una de las limitaciones que afectan este uso de estilo, es la dependencia del servidor, si se llega a presentar falla o una interrupción, todos los clientes quedan sin acceso al sistema lo cuál exige respaldos y monitoreo continuo para garantizar la disponibilidad del servicio y contar con una persona a cargo para llevar a cabo la solución en caso de alguna falla del servidor.

Sobrecarga del servidor: no puede tener gran cantidad de usuarios porque este ralentiza las solicitudes y así afecta el rendimiento, por lo cuál debe contar con una infraestructura adecuada o mecanismos de balanceo de carga o alternos para evitar estas novedades.

Dependencia de la red: El cliente requiere de la conexión constante y disponibilidad del sistema, por ello también necesita del servidor y debe tener conexión constante de lo contrario esto generará una mala experiencia para el usuario o hasta impedir el uso del sistema.

Costos de infraestructura: Debido a que el desempeño con el servidor no es tan eficiente y no cuenta con buena disponibilidad, se debe invertir en una infraestructura robusta o servicios en la nube con escalabilidad dinámica, lo que incrementa los costos.

Seguridad: Cuando se comparte canales de información entre servidores y clientes esto requiere un proceso de validaciones, se debe hacer cumplir con una serie de protocolos de seguridad, para evitar algún tipo de apertura de ataques de malware, amenazas o que generen daños físicos.

• Impacto sobre la escalabilidad, mantenibilidad y desempeño.

Escalabilidad: Este estilo de arquitectura ofrece un alto nivel de escalabilidad especialmente si el servidor se implementa en plataformas en la nube (como AWS,Azure o Google Cloud), así permite manejar el crecimiento del sistema sin necesidad de rediseñar, agregando instancias, contenedores o balanceadores de carga. Esta parte es bastante importante para nuestra aplicación de gestor de tareas porque así podemos manejar de pocos usuarios a un alto número de usuarios activos.

Mantenibilidad: Cuando se tiene la separación de cliente y servidor la arquitectura facilita en gran medida la mantenibilidad del sistema, por lo que la lógica de negocio y el acceso a los datos están concentrados en el servidor, las actualizaciones, las correcciones de errores o mejoras funcionales pueden implementarse directamente en ese componente sin necesidad de modificar o redistribuir las aplicaciones cliente. lo cuál hace que los usuarios finales no se vean afectados y continúan el uso de la misma interfaz mientras el servidor evoluciona internamente. Además, el uso de una API bien definida y documentada permite mantener una comunicación estable entre ambos lados incluso si se agregan nuevas funciones o se optimiza el proceso existente. Gracias a la independencia entre capas reduce el riesgo de introducir fallos colaterales y permite que el equipo de desarrollo trabaja en mejoras continuas de forma controlada, asegurando la evolución sostenida del sistema a largo plazo sin afectar su disponibilidad ni experiencia de uso.

Desempeño: En este estilo el cliente-Servidor, como se había dicho antes el procesamiento principal se concentra en el servidor, donde se ejecutan las operaciones de negocio y se gestionan las consultas a la base de datos. Debido a esta centralización permite un mejor control y el uso preciso de recursos y mejor administración al tráfico de peticiones. Además de hacer la plataforma más amigable con el usuario ya que no deben realizar operaciones complejas, solo se encarga de mostrar la información y enviar solicitudes y las consultas de datos y generación de informes, lo hace el servidor que tiene los recursos necesarios, gracias al uso de mecanismos caché, optimización de consultas SQL y balanceo de carga contribuye a mejorar los tiempos de respuesta y capacidad del sistema para dar respuesta a las múltiples peticiones sin degradar el rendimiento.

Cómo se articula Cliente-Servidor con los entregables posteriores

1) Implementación

Separación de repositorios / carpetas
-client/ → React (UI, rutas, consumo de API, autenticación JWT en cliente).
-server/ → ASP.NET Core Web API (controladores, servicios, repositorios, DTOs).
-infrastructure/ (opcional) → scripts de despliegue, Dockerfiles, Terraform/ARM.

Contrato API (entregable): Documento OpenAPI / Swagger que defina los endpoints, métodos, request/response y códigos HTTP.

Entregable: openapi.yaml o interfaz Swagger publicada.

Gestión de configuración y secretos:

Variables de entorno para cadenas de conexión, JWT secret, y claves de terceros.

Entregable: guía de configuración (README.env.example) y proceso para inyectar secretos en CI/CD.

Migraciones y esquema de BD:

EF Core migrations versionadas en server/Migrations.

Entregable: scripts de migración y guía para ejecutar dotnet ef database update.

Artefactos de compilación:

-Backend: paquete publicado (dotnet publish) o imagen Docker (server/Dockerfile).
-Frontend: build estático (npm run build) que se sirve desde CDN o integrado en el servidor.

Entregable: instrucciones y artefactos (zip/image) listos para despliegue.

2) Despliegue

Estrategia de despliegue (entregable):

Documento que explique el flujo (staging → production), rollback y versiones.

Opciones de despliegue recomendadas:

Contenedores Docker: docker-compose para entorno local / staging; imágenes publicadas en registry (Docker Hub / Azure Container Registry).

PaaS: Azure App Service / AWS Elastic Beanstalk para desplegar el backend; frontend en CDN (Netlify, Vercel, S3+CloudFront).

Servidor tradicional: Kestrel detrás de Nginx como proxy reverso.

CI/CD (ejemplo entregable):

GitHub Actions / GitLab CI pipeline con etapas:

build (client y server)

test (unit + integration)

publish (artefactos o imágenes)

deploy a staging (auto) y a prod (manual/approval)

Entregable: archivo de pipeline (.github/workflows/ci.yml) y diagrama del flujo.

Rollback / Blue-Green / Canary:

Explicar la estrategia elegida (p. ej. despliegue canary o tag + rollback automático si falla healthcheck).

Entregable: procedimiento paso a paso para rollback.

3) Pruebas

Niveles de pruebas (entregables):

Unit tests: para servicios, utilidades, validaciones. (xUnit / NUnit para .NET; Jest para React).

Entregable: cobertura mínima acordada (p. ej. 70%) y reporte de cobertura.

Integration tests: pruebas de API que usan in-memory DB o containers (Testcontainers) para validar contratos.

Entregable: suite de tests de integración automatizada.

End-to-End (E2E): flujos críticos (login, crear/asignar tarea, cerrar tarea) con Cypress o Playwright apuntando a staging.

Entregable: scripts E2E y reporte de resultados.

Pruebas de carga: k6 o Gatling para simular concurrencia y medir latencias.

Entregable: informe de resultados y recomendaciones (puntos de ruptura).

Pruebas de seguridad: escaneo SAST (e.g., SonarCloud) y revisión básica de OWASP (control CSRF, XSS, inyección).

Entregable: resultado del escaneo y lista de remediaciones.

Automatización en CI:Pipeline ejecuta unit+integration en cada PR; E2E y carga en merges a staging o nightly runs.

Entregable: configuración CI que muestre la ejecución automática de tests.

README

Todo App

Parte Backend

La aplicación Todo App es un gestor de tareas, estas permiten crear y cambiar el estado de las tareas a pendiente, en proceso y completado, adicional cuando se crea y asigna una tarea le llega la notificación al usuario por medio de correo, esta aplicación también puede descargar reportes de las tareas en un archivo Excel por parte del usuario con rol supervisor. La aplicación cuenta con un módulo de gestión de usuarios donde el administrador puede realizar creación, actualización y eliminación de usuarios que tengan acceso a esta.

CARACTERÍSTICAS PRINCIPALES

Creaciòn, actualización, y eliminación de tareas

Notificaciòn automática al asignar o actualizar una tarea

Actualizar estado de tareas pendiente, en progreso o completado

Generaciòn de informes Excel

Sistema de autenticaciòn(JWT)

Logs y auditorìa de cambios

Documentaciòn con Swagger

TECNOLOGÍAS UTILIZADAS

ASP.NET+C# (Backend)

.REACT+Vite(Frontend)

Base de datos: SQL Server

Notificaciones: SMT

Autenticaciòn: JSON Web Tokens (JWT)

Archivos de informe: Excel (xlsx)

Vercel (Despliegue Frontend)

Render (Despliegue Backend)

SuperBase (Base de datos en la nube)

ARQUITECTURA UTILIZADA

Cliente servidor: cuando analizamos los diferentes tipos de arquitectura encontramos que la que mejor se adaptó fue cliente servidor por lo que la aplicación tiene su backend y frontend separados y el front (cliente) desde un navegador hace llamado al backend (servidor) para realizar una acción. (procesos de las APIS).

También cuenta con MVC (modelo vista controlador) por la simplicidad de la aplicación el cual en el backend se puede ver los controllers, los (modelos) de datos, DTO (data transfer object), una capa de servicio donde se tienen las integraciones (JWT y notificación por correo), las vistas están en el frontend donde tiene arquitectura basada en componentes y una capa de servicio que llama a las APIS (application programing interface (ESPAÑOL Interfaz de Programación de Aplicaciones)) expuestas en el backend

PATRONES Y SOLID APLICADOS

Singleton se aplicó en el context para el acceso a la base de datos

Inyección dependencia en los controladores de tareas y cuentas inyectando la interfaz de notificación por correo y token

Factory Method, Repository, Command como tal no fueron usados en la aplicación porque no están funcionales y agrega complejidad en la aplicación, pero se dieron como sugerencia para otras versiones futuras.

Los principios SOLID usados son los siguientes

(single responsability) responsabilidad única que se pueden ver en la notificación al correo y en la generación del token

segregación de interfaces como se puede ver en la funcionalidad con autenticación con token y correo, se realizan interfaces correspondientes

Inversión de dependencias donde el controlador de tareas y Cuentas inyectan la dependencia de la interfaz de notificación de correo y generación de token para usar sus métodos.

INSTALACIÒN

Se clona el repositorio

BACKEND https://github.com/DG97-prog/TODOAPI-Backend.git

Abrir proyecto o una solución

Busco el proyecto

TodoApp

Abro carpeta

TodoApp.API.sln

Doble click y abro

Compilar proyecto para descargar dependencias

En la parte superior buscar o crear perfil de ejecución (ejemplo http o Todoapp.http)

Dale ejecutar

Para documentación en swagger colocarle a la URL /swagger

Para que funcione y probar debes tener acceso a la base de datos en la nube o restaurarla en sql server

FLUJO DEL TRABAJO DEL SISTEMA

El administrador es el responsable de

Crear usuario

Actualizar usuario

Eliminar usuario

El administrador puede crear sus propias tareas

El supervisor es el encargado de asignar tareas y puede

Crear

Actualizar

Eliminar

El backend cuando se crea una tarea se encarga de enviar una notificación automática por correo a los usuarios.

El supervisor puede generar informes del listado general de las tareas que incluye (Total tareas por estado, Tareas por usuario, Fecha de creaciòn de la tarea, Fecha de vencimiento)

ESTRUCTURA DEL PROYECTO BACKEND

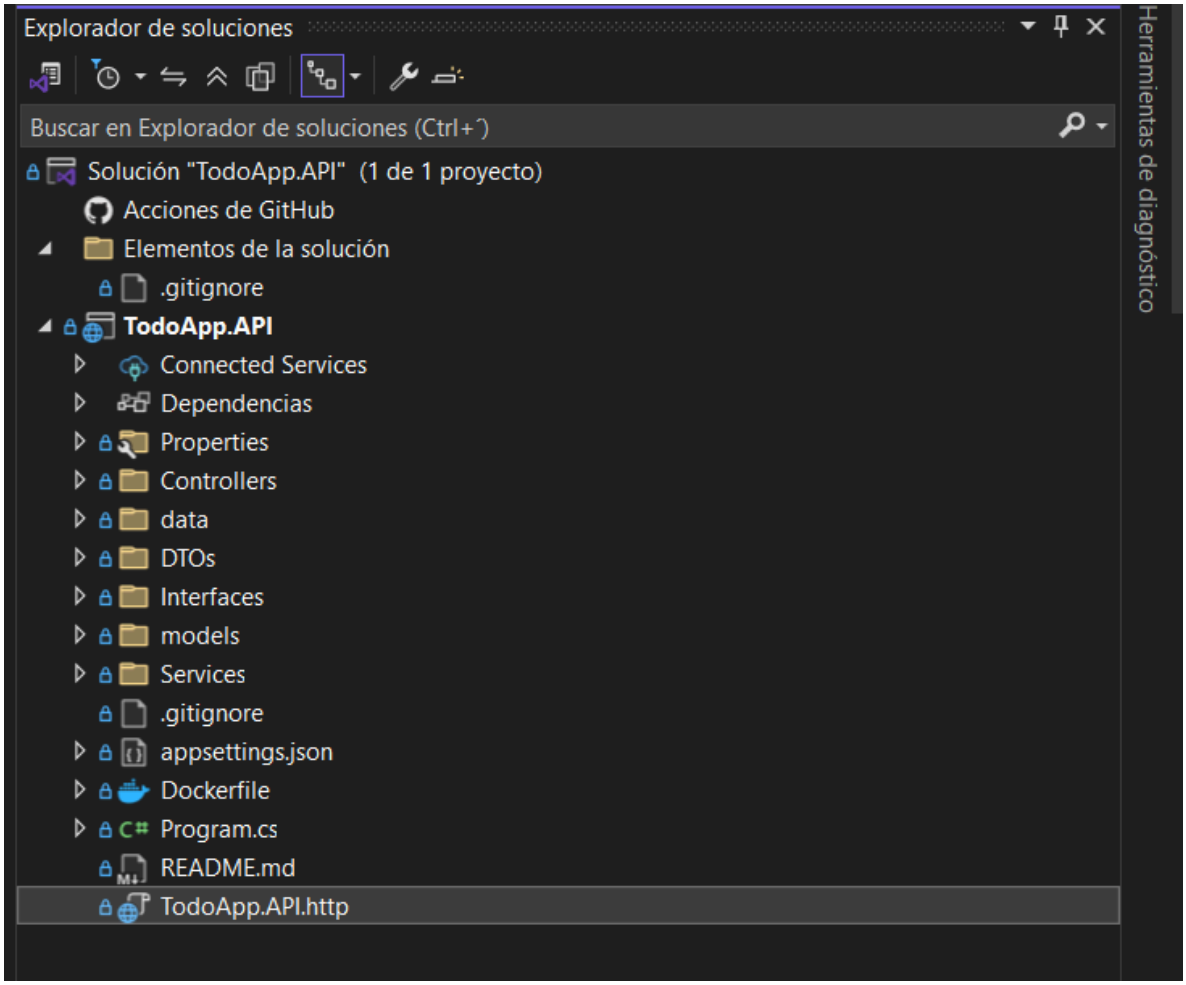
```
TodoApp.API/
│
├── Controllers/
│   ├── AccountController.cs
│   ├── AuthController.cs
│   └── TareasController.cs
│
├── data/
│   └── ApplicationDbContext.cs
│
├── DTOs/
│   ├── CreateTareaDto.cs
│   ├── LoginDto.cs
│   ├── RegisterDto.cs
│   ├── UpdateTareaDto.cs
│   └── UpdateUsuarioDto.cs
│
├── Interfaces/
│   ├── IAuthService.cs
│   └── IEmailService.cs
```

```
│
├── models/
│   ├── Categoria.cs
│   ├── Estado.cs
│   ├── Tarea.cs
│   └── Usuario.cs
│
├── Services/
│   ├── AuthService.cs
│   └── EmailService.cs
│
└── Properties/
```

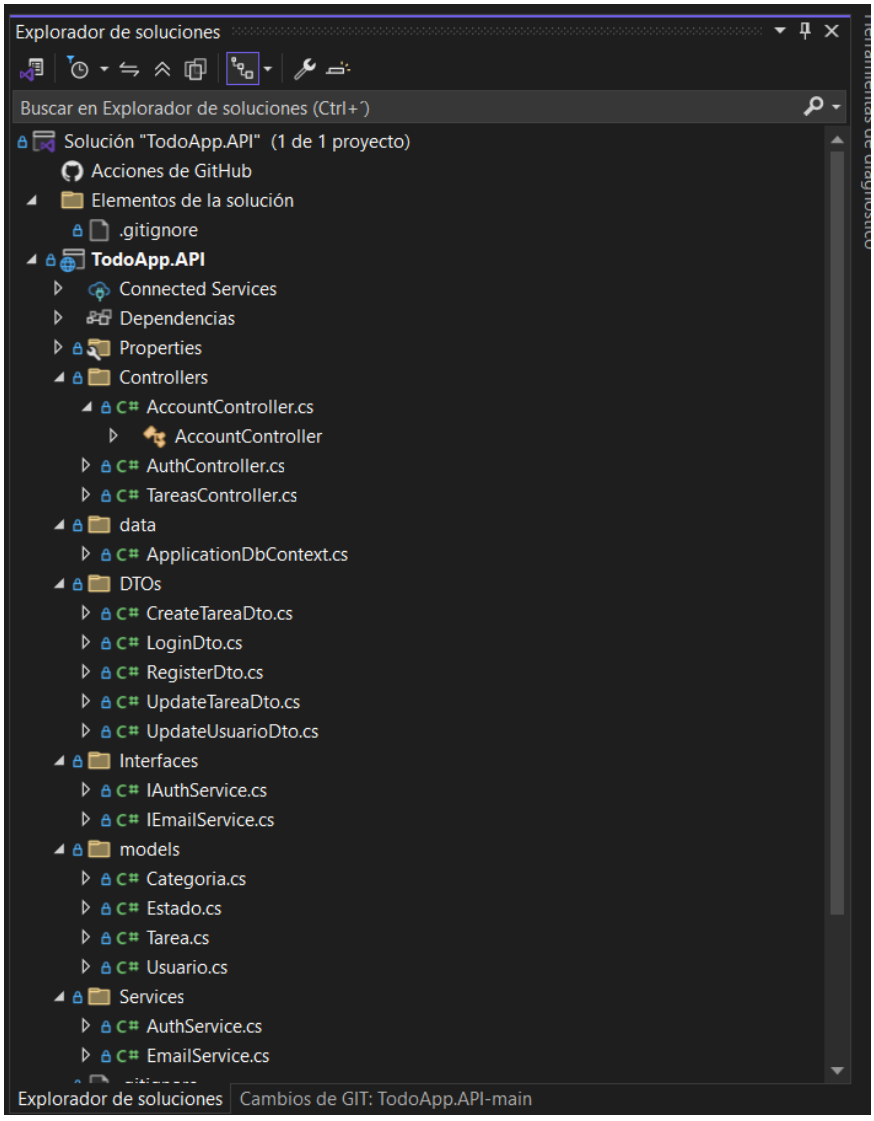
SCREENSHOTS O EVIDENCIAS

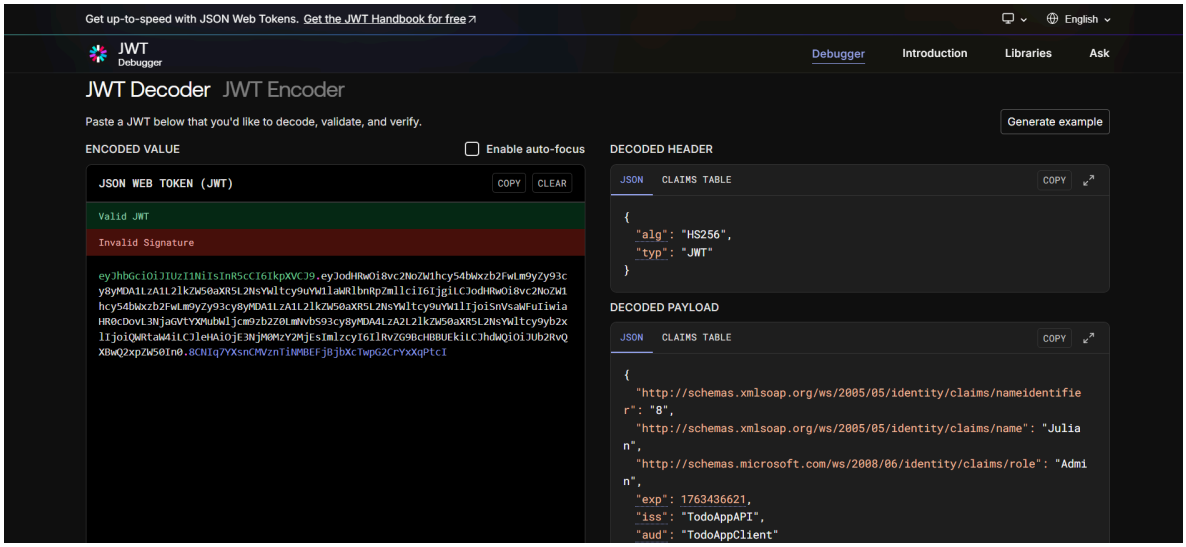
A continuación se muestran las principales vistas de la aplicación.

Estructura código, podemos ver las diferentes

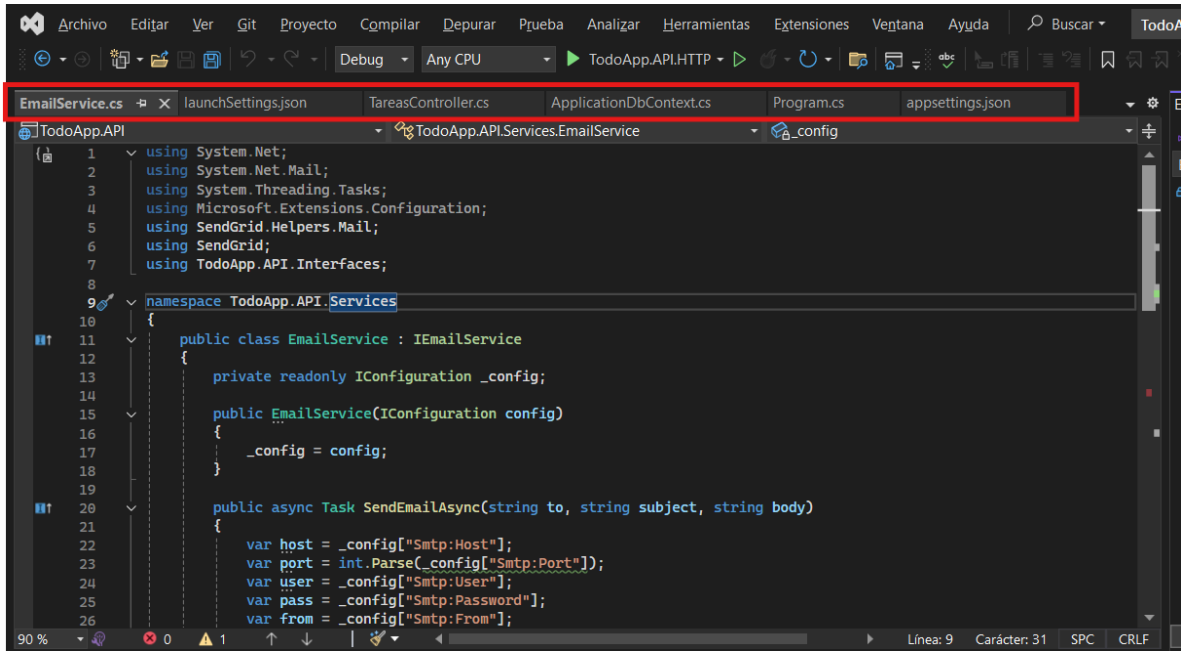

Explorador de soluciones
Buscar en Explorador de soluciones (Ctrl+´)
Solución "TodoApp.API" (1 de 1 proyecto)
Acciones de GitHub
Elementos de la solución
.gitignore
TodoApp.API
Connected Services
Dependencias
Properties
Controllers
data
DTOs
Interfaces
models
Services
.gitignore
appsettings.json
Dockerfile
Program.cs
README.md
TodoApp.API.http
Herramientas de diagnóstico

Estructura carpetas

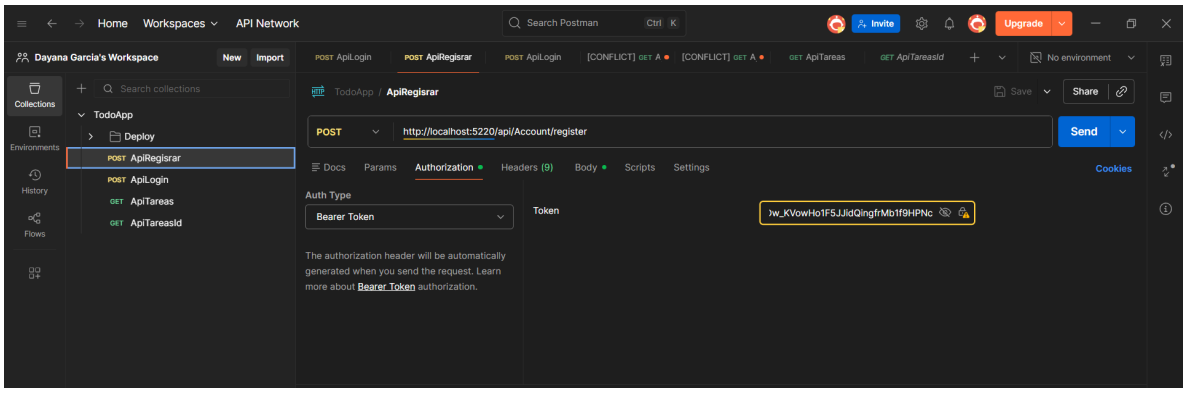



Get up-to-speed with JSON Web Tokens. Get the JWT Handbook for free
English
JWT Debugger
Debugger
Introduction
Libraries
Ask
JWT Decoder
JWT Encoder
Paste a JWT below that you'd like to decode, validate, and verify.
Generate example
ENCODED VALUE
Enable auto-focus
JSON WEB TOKEN (JWT)
COPY
CLEAR
Valid JWT
Invalid Signature
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWRlbnRpZmllciI6IjgiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoiSnVsaWFuIiwiaHR0cDovL3NjaGVtYXMubWljcm9zb2Z0LmNvbS93cy8yMDA4LzA2L2lkZW50aXR5L2NsYWltcy9yb2xlIjoiQWRtaW4iLCJleHAiOjE3NjM0MzY2MjEsImlzcyI6IlRvZG9BcHBBUEkiLCJhdWQiOiJUb2RvQXBwQ2xpZW50In0.8CNIq7YXsnCMVznTiNMBEFj0jbXcTwpG2CrYxXqPtcI
DECODED HEADER
JSON
CLAIMS TABLE
COPY
{
"alg": "HS256",
"typ": "JWT"
}
DECODED PAYLOAD
JSON
CLAIMS TABLE
COPY
{
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier": "8",
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name": "Julian",
"http://schemas.microsoft.com/ws/2008/06/identity/claims/role": "Admin",
"exp": 1763436621,
"iss": "TodoAppAPI",
"aud": "TodoAppClient"

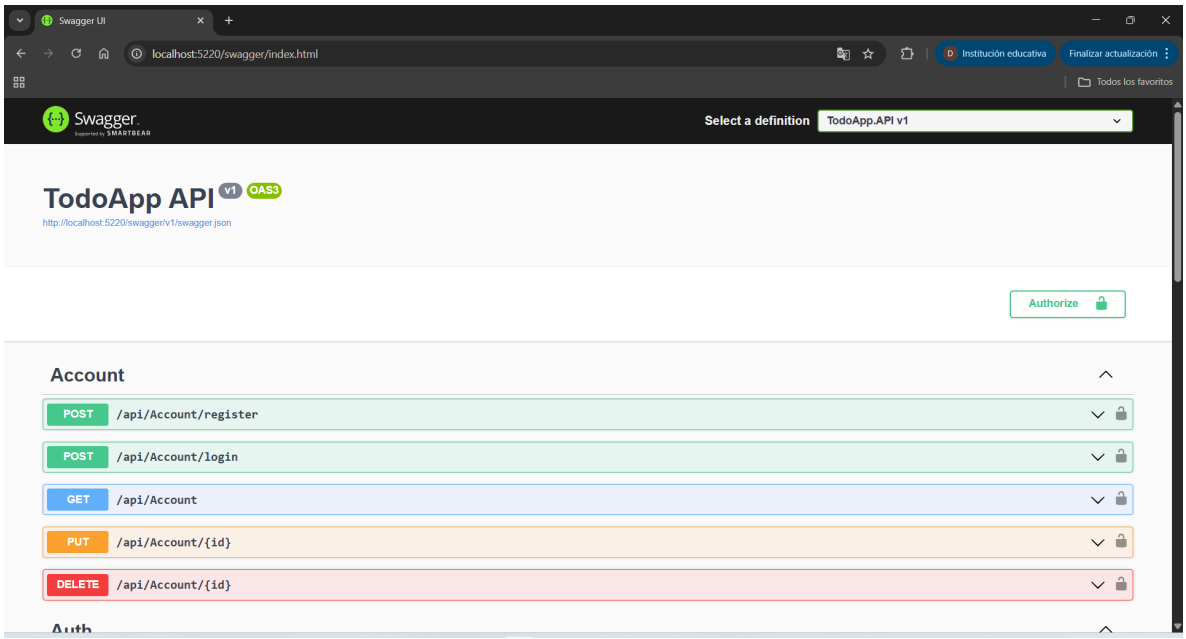
Código


Archivo Editar Ver Git Proyecto Compilar Depurar Prueba Analizar Herramientas Extensiones Ventana Ayuda Buscar
Debug
Any CPU
TodoApp.API.HTTP
EmailService.cs
launchSettings.json
TareasController.cs
ApplicationDbContext.cs
Program.cs
appsettings.json
TodoApp.API
TodoApp.API.Services.EmailService
_config
using System.Net;
using System.Net.Mail;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using SendGrid.Helpers.Mail;
using SendGrid;
using TodoApp.API.Interfaces;
namespace TodoApp.API.Services
{
public class EmailService : IEmailService
{
private readonly IConfiguration _config;
public EmailService(IConfiguration config)
{
_config = config;
}
public async Task SendEmailAsync(string to, string subject, string body)
{
var host = _config["Smtp:Host"];
var port = int.Parse(_config["Smtp:Port"]);
var user = _config["Smtp:User"];
var pass = _config["Smtp:Password"];
var from = _config["Smtp:From"];
90 %
0
1
Línea: 9
Carácter: 31
SPC
CRLF

Valdicación funcionamiento de las APIS con postman



Ejeccución aplicación Swagger

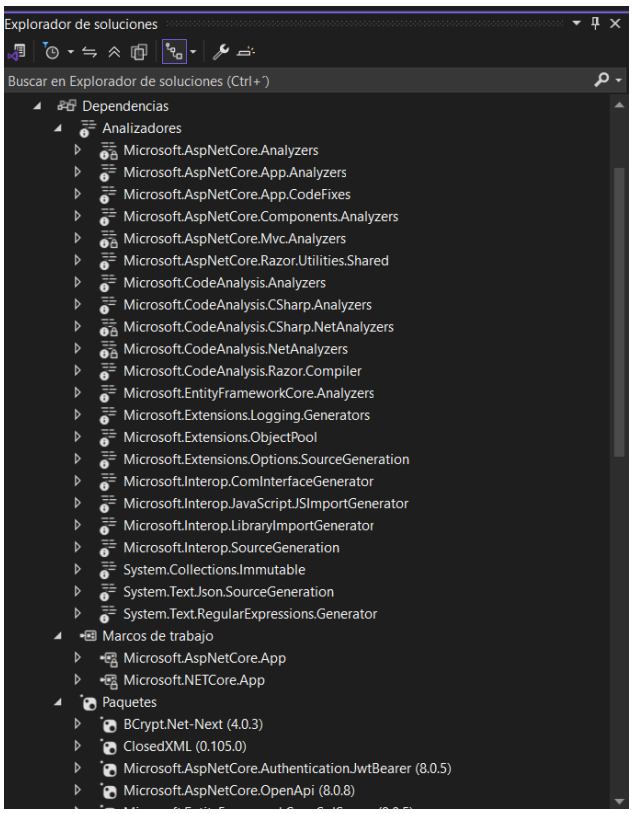


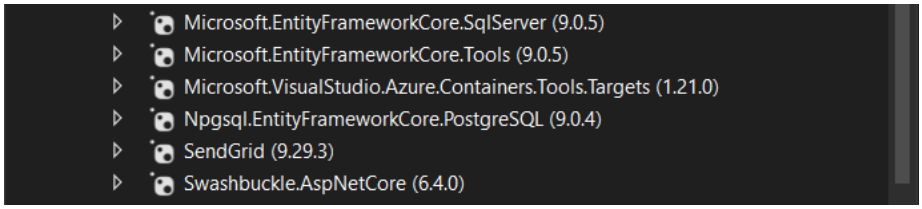
HERRAMIENTAS EXTERNAS/OTRAS DEPENDENCIAS

JWT para la autenticación de la creación de los usuarios.

Notificación por correo (SMT)

Swagger





Parte Frontend

Todo App

Todo App es la interfaz de usuario, supervisor y administrador donde los diferentes miembros según su usuario asignado del equipo de trabajo pueden administrar sus tareas de manera simple y dinámica.

Se conecta al backend mediante una API para mostrar tareas, asignaciones, estados y notificaciones según el usuario, a través de un panel interactivo, donde se puede ver el listado de las tareas asignadas y descargar informes generales en archivo xlsx.

CARACTERÍSTICAS PRINCIPALES

Panel intuitivo y dinámico.

Vista Kanban (Pendiente → En progreso → Completado).

Gestión de usuarios desde la parte de admi (login, registro) puede crear los usuarios y crear sus tareas personales,pero no asignarlas a otros miembros del equipo.

Supervisor (login) puede crear tareas y asignarlas a otros miembros del equipo de trabajo.

Usuario (login) puede crear tareas personales, pero no asignarlas a otros miembros del equipo de trabajo.

Actualización automática del estado de tareas.

Notificaciones visuales y en tiempo real a los usuarios.

Generación y descarga de informes desde la interfaz.

Integraciòn completa con la API del backend

TECNOLOGÌAS UTILIZADAS

Framework: Vite + React

Estilos: CSS, lucide react

Construcciòn: Vite

Autenticaciòn: JWT

Comunicaciòn: Fetch API

Notificaciones: SMTP

Estado global: useState

Vercel (Despliegue Frontend)

INSTALACIÒN

Clonar el repositorio

FRONTEND https://github.com/DG97-prog/TODOAPI-FrontEnd.git

Se abre visual studio code

Se abre la carpeta de la app TodoApp Main main

Cuando abra la aplicación realizar npm i para descargar dependencias (ver Package.json)

en terminal (bash si es Windows)

Después de descargar dependencias en la terminal ejecutar

npm run dev

Te mostrará una URL la cual la usas en cualquier navegador para interactuar con la aplicación

ESTRUCTURA DEL PROYECTO FRONTEND

TODO-APP/

│

├── node_modules/

│

```
├── public/
│
├── src/
│   ├── assets/
│   │
│   ├── components/
│   │   ├── Login.jsx
│   │   ├── TaskForm.jsx
│   │   ├── TaskItem.jsx
│   │   ├── TaskStats.jsx
│   │   └── UserAdmin.jsx
│   │
│   ├── services/
│   │   ├── apiService.js
│   │   └── mockService.js
│   │
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
│
```

├── .gitignore

├── eslint.config.js

├── index.html

├── package-lock.json

├── package.json

├── postcss.config.js

├── README.md

├── tailwind.config.js

└── vite.config.js

FUNCIONALIDADES DE LA INTERFAZ

Panel de Usuario

Lista de tareas asignadas

Total de tareas

Total tareas segùn el estado

Vista Kanban

Estados:

-Pendiente

-En progreso

-Completada

Panel de Informes

Botòn para descargar informes generados por el backend

Los informes incluirán

Total tareas por estado

Tareas por usuario

Fecha de creaciòn de la tarea

Fecha de vencimiento

Gestiòn de roles y permisos en la interfaz

Administrador

-Acceso completo al panel

-Gestionar usuarios (crear, elimina y actualiza)

-Crear, editar y eliminar tareas personales

-Recibir notificaciones por correo cuando le asignan una tarea o cuando la crea

Supervisor

-Acceso completo al panel

-Crear, editar y eliminar tareas personales

-Descargar informe de las tareas de todo el equipo en archivo xlsx

-No puede modificar roles ni eliminar usuarios

-Recibir notificaciones por correo cuando le asignan una tarea o cuando la crea

Usuario

-Acceso completo al panel

-Crear, editar y eliminar tareas personales

-No puede modificar roles ni eliminar usuarios

-Recibir notificaciones por correo cuando le asignan una tarea o cuando la crea

AUTENTICACIÒN

El frontend utiliza JWT proporcionado por el backend.

El token se almacena en:

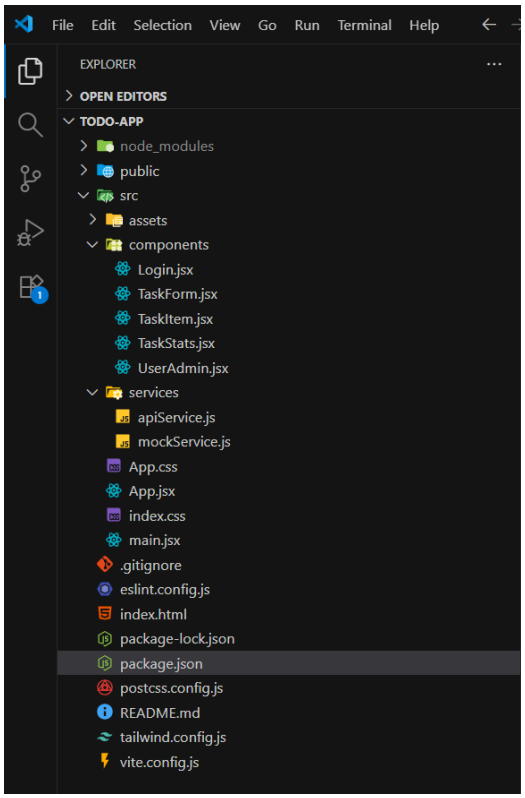
LocalStorage / SesionStorage

Interceptores de Axios para cada peticiòn

Se renueva automáticamente cuando es necesario (si está implementado en backend)

SCREENSHOTS O EVIDENCIAS

Estructura carpetas



HERRAMIENTAS EXTERNAS/OTRAS DEPENDENCIAS

package.json > {} scripts

```
{
  "name": "todo-app",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  ▷Debug
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview"
  },
  "dependencies": {
    "lucide-react": "^0.511.0",
    "react": "^19.1.0",
    "react-dom": "^19.1.0"
  },
  "devDependencies": {
    "@eslint/js": "^9.25.0",
    "@tailwindcss/postcss": "^4.1.7",
    "@types/react": "^19.1.2",
    "@types/react-dom": "^19.1.2",
    "@vitejs/plugin-react": "^4.4.1",
    "autoprefixer": "^10.4.21",
    "eslint": "^9.25.0",
    "eslint-plugin-react-hooks": "^5.2.0",
    "eslint-plugin-react-refresh": "^0.4.19",
    "globals": "^16.0.0",
    "postcss": "^8.5.3",
    "tailwindcss": "^3.4.17",
    "vite": "^6.3.5"
  }
}
```

Enlace para ingresar a la aplicación en despliegue

https://todoapi-front-end.vercel.app/

Tabla de imágenes