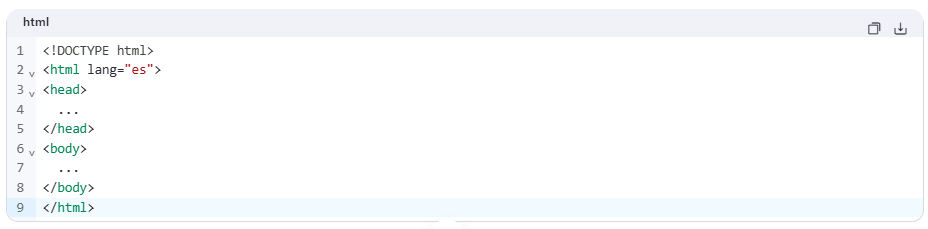
**Explicación Detallada del Archivo *index.html* de TaskFlow**

Este archivo es el corazón del frontend de tu aplicación TaskFlow. Es el primer archivo que carga el navegador y contiene toda la estructura visual de la aplicación. A continuación, te explico cada parte de forma clara y detallada.

**🏗️ Estructura General del HTML**



* <!DOCTYPE html>: Define que este es un documento HTML5.
* <html lang="es">: Indica que el contenido está en español.
* <head>: Contiene metadatos, títulos y enlaces a recursos externos (CSS, fuentes).
* <body>: Contiene todo el contenido visible de la página.

**🔤 Sección** <head>**: Información y Recursos**

**<meta charset="UTF-8" />** 🡪 Define que el documento usa codificación de caracteres UTF-8 (incluye tildes, ñ, etc.).

**<meta name="viewport" content="width=device-width, initial-scale=1.0"/> 🡪** Hace que la página sea responsive (se adapte a móviles y tablets).

**<title>TaskFlow - Gestor de Tareas</title> 🡪** Título que aparece en la pestaña del navegador.

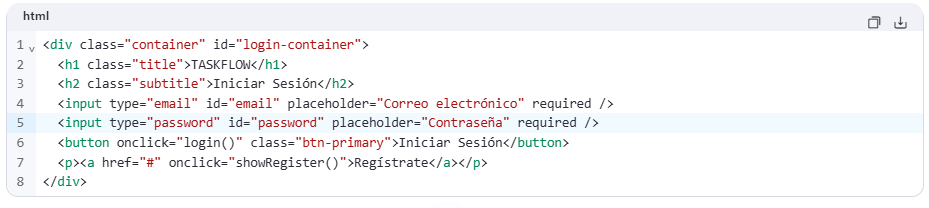
**<link href="https://fonts.googleapis.com/css2?family=Press+Start+2P&display=swap" rel="stylesheet"> 🡪** Carga la fuente "Press Start 2P", un estilo pixelado y retro, perfecto para el diseño anime/cyberpunk.

**<link rel="stylesheet" href="style.css" /> 🡪** Enlaza el archivo style.css que contiene todos los estilos visuales (colores, animaciones, diseño).

**🖥️ Sección <body>: Las Tres Pantallas del Sistema**

El cuerpo de la página tiene tres "pantallas" que se muestran u ocultan según el estado del usuario:

**1. 📱 Pantalla de Login**

****

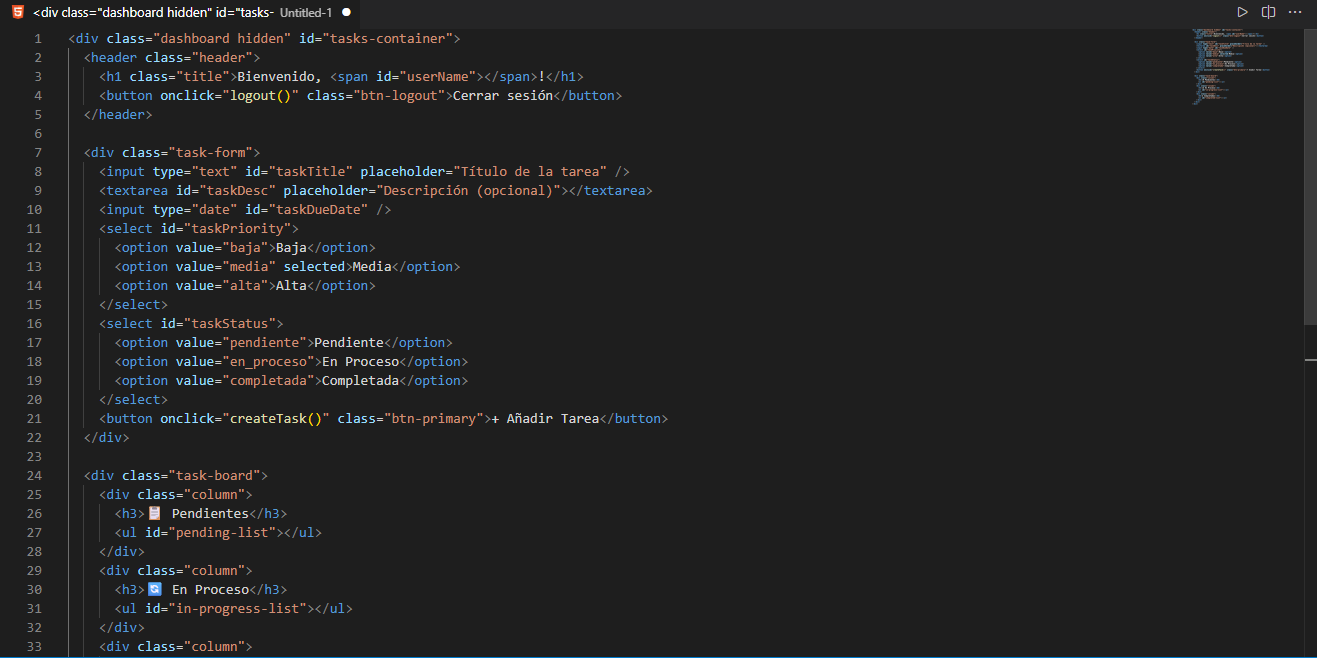
* **class="container":** Define un contenedor estilizado (fondo oscuro, bordes neón).
* **id="login-container":** Identificador único para controlar esta pantalla desde JavaScript.
* **<input type="email"> y <input type="password">:** Campos para correo y contraseña.
* **required:** Obliga al usuario a llenar el campo.
* **onclick="login()":** Al hacer clic, ejecuta la función login() del archivo script.js.
* **<a href="#" onclick="showRegister()">:** Enlace que cambia a la pantalla de registro.

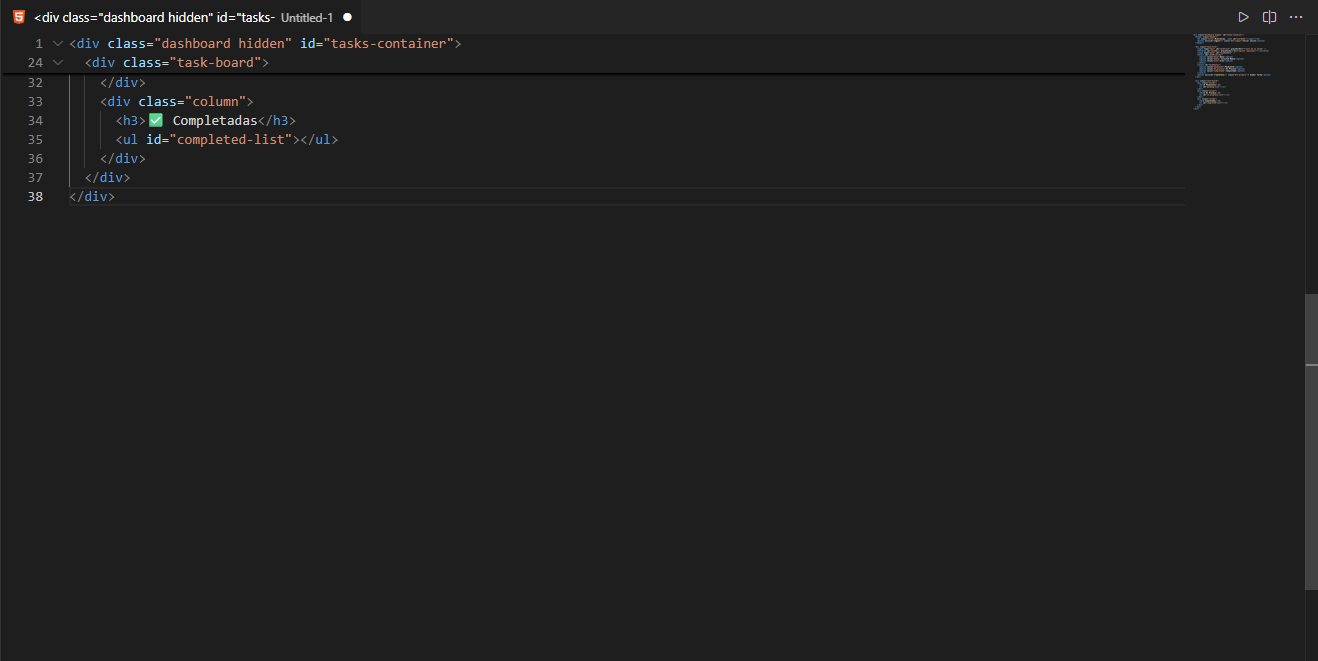
**2. ✍️ Pantalla de Registro**

****

* **class="hidden":** Esta pantalla está oculta al inicio.
* **id="register-container":** Para controlarla con JavaScript.
* **Campos:** Nombre, correo y contraseña.
* **onclick="register()":** Ejecuta la función register() del script.js.
* **onclick="showLogin()":** Vuelve a la pantalla de login.

**3. 📋 Dashboard de Tareas (Principal)**

****

****

**a) Header del Dashboard**

* **<span id="userName"></span>:** Aquí se mostrará el nombre del usuario cuando inicie sesión.
* **onclick="logout()":** Cierra la sesión y vuelve al login.

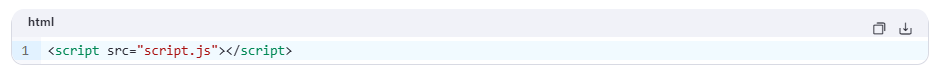
**b) Formulario de Nueva Tarea (task-form)**

* **id="taskTitle":** Título de la tarea.
* **id="taskDesc":** Descripción opcional.
* **id="taskDueDate":** Selector de fecha de vencimiento.
* **<select id="taskPriority">:** Prioridad (baja, media, alta).
* **<select id="taskStatus">:** Estado de la tarea.
* **onclick="createTask()":** Crea una nueva tarea (función en script.js).

**c) Tablero de Tareas (task-board)**

* **Tres columnas:**
  1. **Pendientes:** Tareas que no han empezado.
  2. **En Proceso:** Tareas que se están trabajando.
  3. **Completadas:** Tareas finalizadas.
* **<ul id="...">:** Cada lista (ul) contendrá las tareas dinámicas generadas por JavaScript.

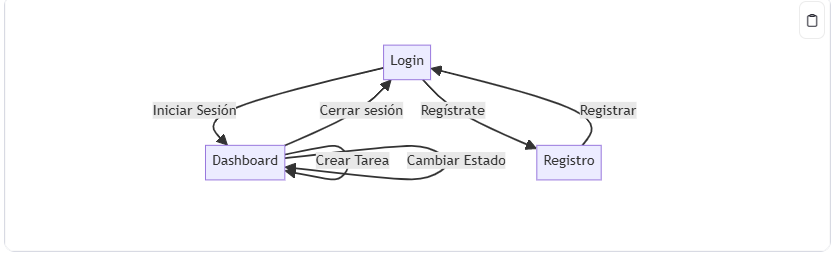
**📜 Carga del JavaScript**

****

* Este comando carga el archivo **script.js**, que contiene toda la **lógica del frontend:**
  + Autenticación (login/register)
  + CRUD de tareas
  + Manejo de estados y prioridades
  + Comunicación con el backend

**⚠️ Importante:** Este script se carga al final del **<body>** para asegurar que todo el HTML esté cargado antes de ejecutar el JavaScript.

**🎨 Resumen Visual del Flujo**

****

**✅ Conclusión**

Este archivo **index.html** es **más que una simple página web.** Es una **aplicación completa** con:

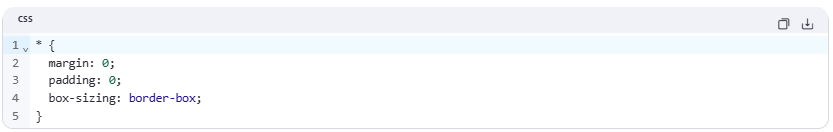
* **✅** Diseño moderno y estilo anime/cyberpunk
* ✅ Navegación entre pantallas sin recargar
* **✅** Formularios interactivos
* ✅ Integración con JavaScript para funcionalidad dinámica
* ✅ Estructura clara y profesional

Es la **base visual** sobre la que se construye todo el sistema TaskFlow.

**🎨 Explicación Detallada del Archivo *style.css* de TaskFlow**

Este archivo **style.css** es el **corazón del diseño visual** de tu aplicación **TaskFlow.** Define cómo se ve cada elemento de la página: colores, formas, animaciones y comportamiento en diferentes dispositivos. A continuación, te explico cada parte de forma clara y detallada.

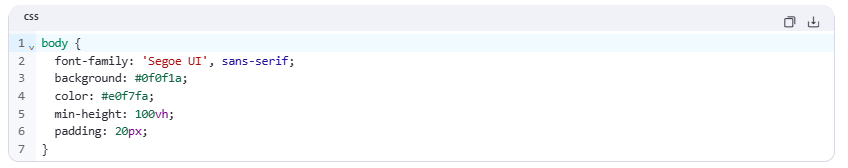
**🧱 Sección 1: Reset y Configuración Inicial**

****

* **\*:** Aplica esta regla a todos los elementos de la página.
* **margin: 0; padding: 0;:** Elimina los márgenes y rellenos predeterminados que los navegadores añaden.
* **box-sizing: border-box;:** Hace que el ancho de un elemento incluya el padding y el border, lo que facilita el diseño responsive.

**✅** Esto asegura que la página se vea igual en todos los navegadores.

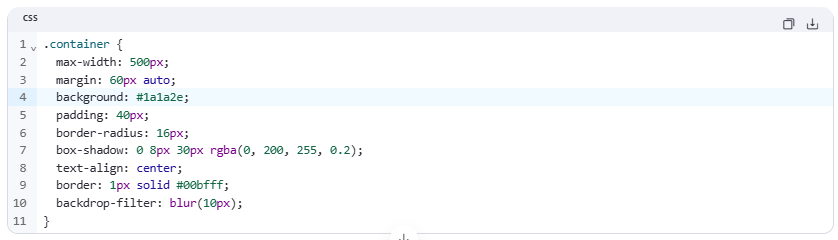
**🌌 Sección 2: Estilo del Cuerpo (body)**

****

* **font-family:** Usa la fuente "Segoe UI", limpia y moderna.
* **background: #0f0f1a;:** Fondo oscuro (casi negro con tono azul profundo).
* **color: #e0f7fa;:** Color del texto (celeste claro, fácil de leer sobre fondo oscuro).
* **min-height: 100vh;:** Asegura que el fondo cubra toda la pantalla.
* **padding: 20px;:** Espacio alrededor del contenido.

**🎨** Este estilo crea una atmósfera nocturna y profesional, ideal para el modo oscuro.

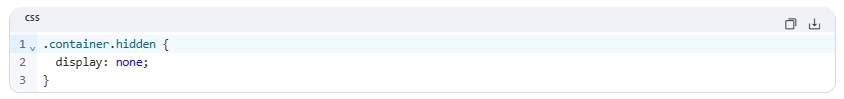
**📦 Sección 3: Contenedores (container)**

****

* **max-width: 500px;:** Ancho máximo del contenedor.
* **margin: 60px auto;:** Centra el contenedor horizontalmente.
* **background: #1a1a2e;:** Fondo oscuro azulado.
* **border-radius: 16px;:** Bordes redondeados.
* **box-shadow:** Sombra neón azul.
* **border: 1px solid #00bfff;:** Borde azul brillante.
* **backdrop-filter: blur(10px);:** Efecto de vidrio esmerilado (como en iOS).

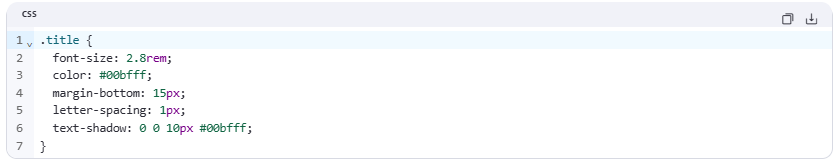
**✨** Este estilo da un efecto **moderno y premium** a las pantallas de login y registro.

**Ocultar Contenedores**

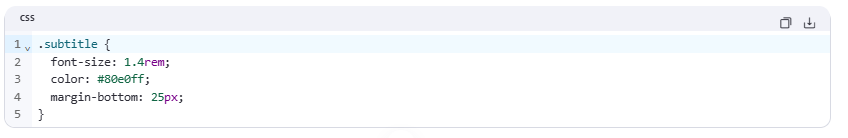
****

* Cuando un contenedor tiene la clase hidden, **se oculta completamente.**
* Esto permite cambiar entre pantallas (login ↔ registro) sin recargar la página.

**🏷️ Sección 4: Títulos (title y subtitle)**

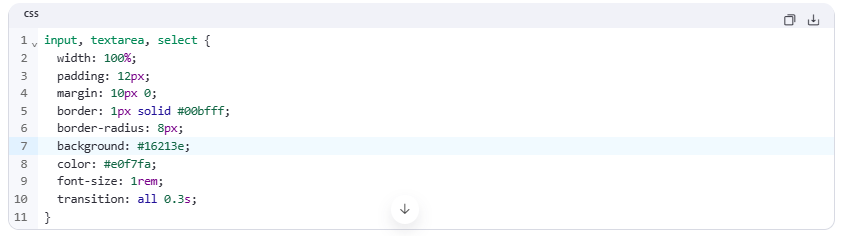
****

* **font-size: 2.8rem;:** Muy grande, impactante.
* **color: #00bfff;:** Azul neón.
* **text-shadow: 0 0 10px #00bfff;:** Brillo alrededor del texto (efecto neón).

****

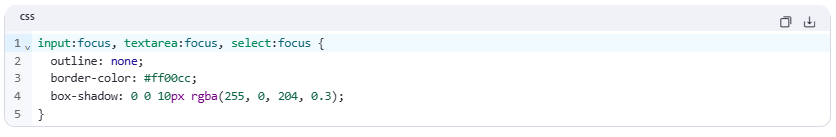
* Subtítulo más pequeño, en tono celeste.

**🔧 Sección 5: Formularios (input, textarea, select)**

****

* **width: 100%:** Ocupan todo el ancho del contenedor.
* **background: #16213e;:** Fondo azul oscuro.
* **border: 1px solid #00bfff;:** Borde azul brillante.
* **transition: all 0.3s;:** Animaciones suaves al interactuar.

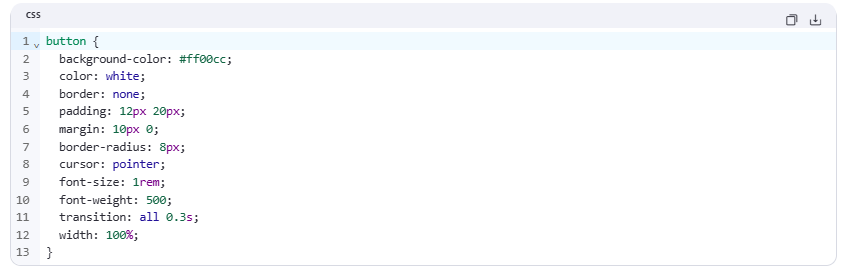
**Enfocar (Focus)**

****

* Al hacer clic en un campo:
  + **outline: none:** Elimina el borde predeterminado.
  + **border-color: #ff00cc:** Cambia a rosa neón.
  + **box-shadow:** Brillo rosa alrededor.

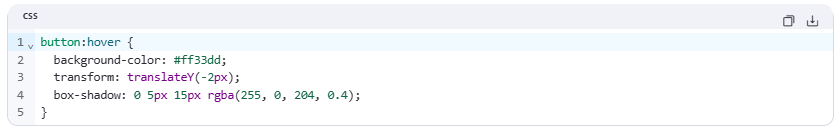
**✨** Efecto visual que indica al usuario que está interactuando con el campo.

**🎯 Sección 6: Botones**

****

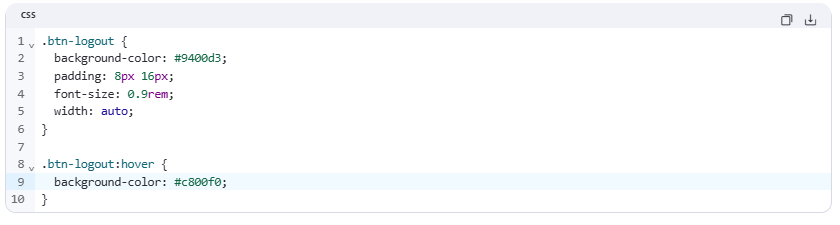
* **background: #ff00cc:** Fondo rosa neón.
* **border: none:** Sin borde (el estilo viene del fondo y radio).
* **cursor: pointer:** Cambia el cursor al pasar el mouse.
* **width: 100%:** Ocupa todo el ancho.

**Hover (Al pasar el mouse)**

****

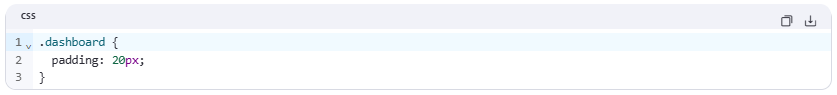
* **background-color: #ff33dd:** Rosa más claro.
* **transform: translateY(-2px):** Sube el botón 2px (efecto de elevación).
* **box-shadow:** Sombra rosa para profundidad.

**Botón de Cerrar Sesión**

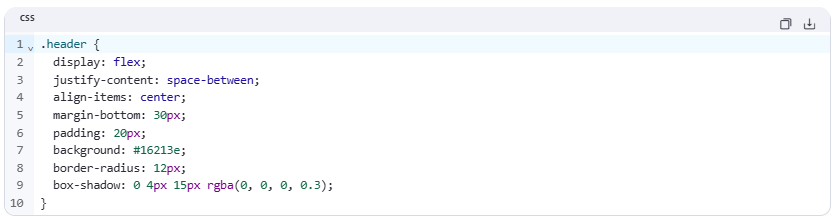
****

* **#9400d3:** Morado oscuro.
* **width: auto:** Solo ocupa el espacio necesario.
* **Hover:** Cambia a morado claro.

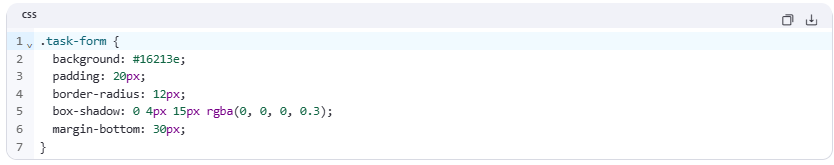
**📊 Sección 7: Dashboard y Formulario de Tareas**

****

* Espacio alrededor del dashboard.

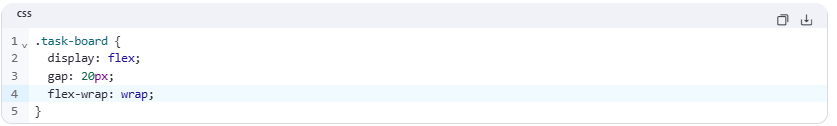


* **display: flex:** Coloca el título y el botón de logout en línea.
* **justify-content: space-between:** Los separa a los extremos.
* Fondo azul oscuro con sombra.

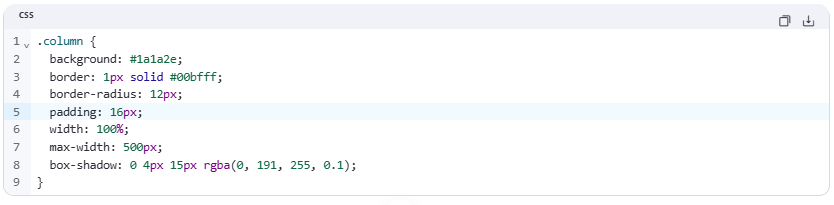


* Formulario para crear tareas, con estilo consistente.

**🗂️ Sección 8: Tablero de Tareas (**task-board**)**

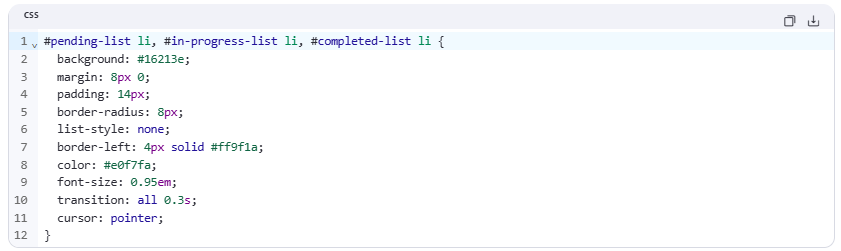


* **display: flex:** Las columnas se colocan en fila.
* **gap: 20px:** Espacio entre columnas.
* **flex-wrap: wrap:** Si no hay espacio, las columnas bajan.

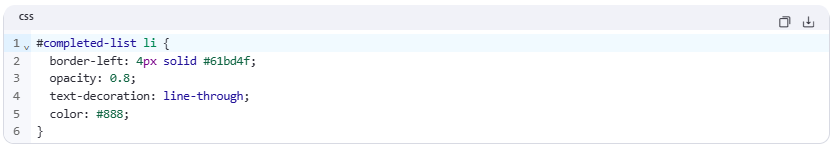


* Cada columna tiene fondo oscuro, borde azul y sombra sutil.

**📝 Sección 9: Listas de Tareas**

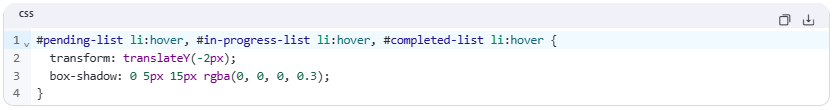


* **border-left: 4px solid #ff9f1a;:** Línea naranja a la izquierda (para tareas pendientes).
* **cursor: pointer:** Indica que se puede interactuar.



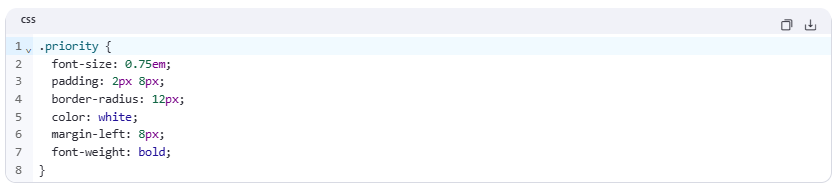
* **Tareas completadas:**
  + Borde verde.
  + Texto tachado.
  + Opacidad reducida (parece "desvanecido").

**Hover en Tareas**

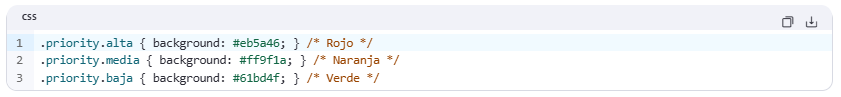
****

* Al pasar el mouse, la tarea **sube ligeramente y gana sombra**, como si flotara.

**🏷️ Sección 10: Etiquetas de Prioridad**

****

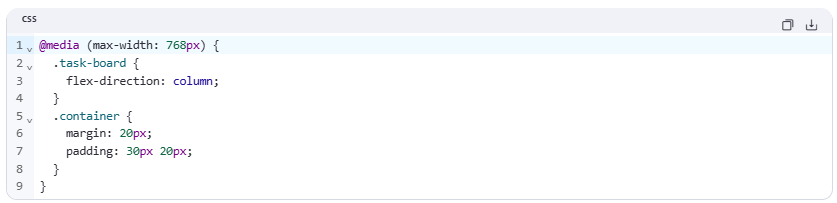
* Estilo base para todas las etiquetas de prioridad.



* Cada prioridad tiene un color distinto:
  + **Alta:** Rojo (urgente)
  + **Media:** Naranja
  + **Baja:** Verde (tranquilo)

✅ Esto permite identificar rápidamente el nivel de urgencia.

**📱 Sección 11: Diseño Responsive (Para Móviles)**

****

* **@media:** Reglas que se aplican solo en pantallas pequeñas.
* **flex-direction: column:** Las columnas de tareas se apilan una debajo de otra.
* **.container:** Menos margen y padding para aprovechar mejor el espacio.

**📱** Esto asegura que la app se vea bien en móviles y tablets.

**✅ Conclusión**

Este archivo **style.css** es **más que un simple archivo de estilos**. Es una **obra de arte digital** que combina:

* **✅ Diseño moderno** con colores neón y efectos visuales.
* **✅ Experiencia de usuario** con animaciones suaves y retroalimentación visual.
* **✅** **Responsividad** para funcionar en cualquier dispositivo.
* **✅ Consistencia visual** en todos los elementos.

Es el alma estética de **TaskFlow**, transformando un sistema funcional en una **aplicación atractiva y profesional.**

**📄 Explicación Detallada del Archivo *script.js* de TaskFlow**

Este archivo **script.js** es el **corazón del frontend** de tu aplicación **TaskFlow**. Es el código que hace que todo funcione: permite el login, registro, creación de tareas y la interacción con el backend. A continuación, te explico cada función de forma clara y detallada.

**🔧 1. showRegister() y showLogin(): Cambiar entre pantallas**

****

* **¿Qué hacen?**Cambian entre la pantalla de login y registro sin recargar la página.
* **document.getElementById():** Obtiene un elemento del HTML por su id.
* **.classList.add('hidden'):** Añade la clase hidden (definida en style.css) para ocultar el contenedor.
* **.classList.remove('hidden'):** Quita la clase hidden para mostrar el contenedor.

**✅** Esto crea una experiencia de usuario fluida, como en apps modernas.

**🔐 2. login(): Iniciar Sesión**

****

**Paso a Paso:**

1. **Obtiene los datos del formulario:**
   * **email** y **password** desde los campos del HTML.
2. **Envía una petición al backend:**
   * **fetch('/api/auth/login'):** Llama a la ruta de login.
   * **method: 'POST':** Envía datos.
   * **headers:** Indica que el contenido es JSON.
   * **body:** Los datos en formato JSON (**{ email, password }**).
3. **Recibe la respuesta:**
   * Si **res.ok** (200-299), significa que el login fue exitoso.
   * Guarda el **token JWT** y el **nombre del usuario** en **localStorage** (como una memoria local del navegador).
4. **Cambia a la pantalla de tareas:**
   * **loadTasks():** Carga las tareas del usuario.
   * **showTasks():** Muestra el dashboard.
5. **Si falla:**
   * Muestra un mensaje de error con **alert()**.

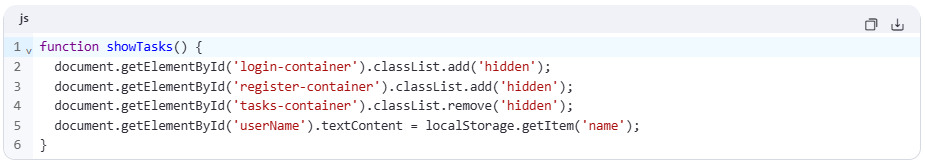
**🔒** El token es esencial para acceder a rutas protegidas como **/api/tasks.**

**📝 3. register(): Registrar un Nuevo Usuario**

****

* Similar a **login()**, pero llama a **/api/auth/register**.
* Envía **name**, **email** y **password**.
* Si tiene éxito, muestra un mensaje y redirige al login.

**📋 4. showTasks(): Mostrar el Dashboard**

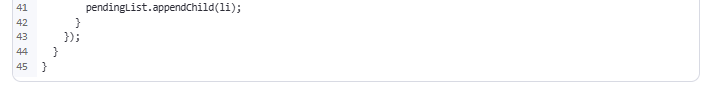
****

* Oculta las pantallas de login y registro.
* Muestra el dashboard (**tasks-container**).
* Muestra el nombre del usuario en el saludo: *"Bienvenido, [nombre]!"*.

**🔄 5. loadTasks(): Cargar las Tareas del Usuario**

****

****

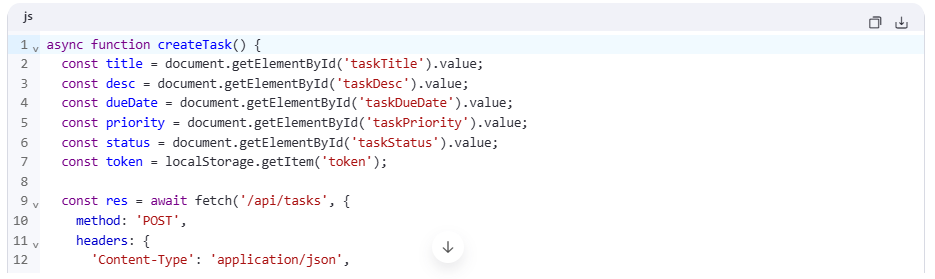
****

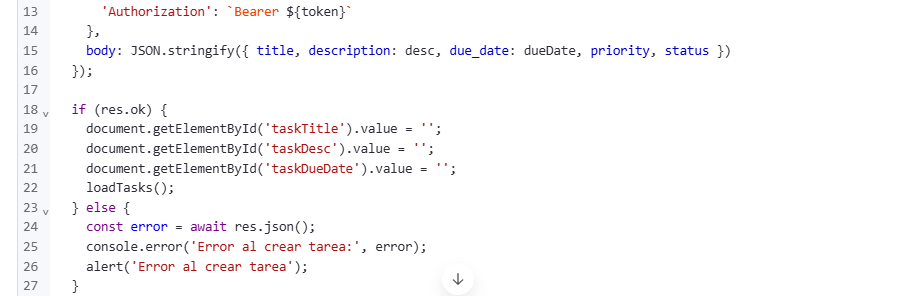
**Paso a Paso:**

1. **Obtiene el token** del **localStorage**.
2. **Llama a /api/tasks** con el token en el encabezado **Authorization**.
3. **Si la respuesta es exitosa:**
   * Obtiene las tareas en formato JSON.
   * Limpia las listas (**innerHTML = ''**) para evitar duplicados.
4. **Para cada tarea:**
   * Crea un elemento **<li>** (una tarjeta).
   * Añade una etiqueta de **prioridad** (baja, media, alta) con color según el valor.
   * Muestra la **fecha de vencimiento** si existe**.**
   * Añade un **<select>** para cambiar el estado.
   * Añade un botón para eliminar la tarea.
5. **Coloca la tarea en la columna correcta:**
   * Pendientes, En Proceso o Completadas.

**✅** Así el usuario ve sus tareas organizadas como en Trello.

**➕ 6. createTask(): Crear una Nueva Tarea**

****

****

****

* Recoge todos los datos del formulario.
* Llama a **/api/tasks** con método **POST**.
* Si tiene éxito, limpia el formulario y recarga las tareas.
* Si falla, muestra un error.

**🔄 7. updateTaskStatus(id, status): Cambiar el Estado de una Tarea**

****

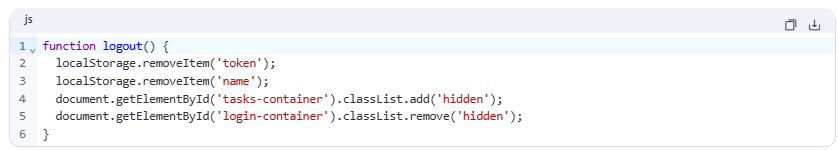
* Llama a **/api/tasks/:id/status** con método **PUT**.
* Actualiza el estado de la tarea.
* Recarga las tareas para reflejar el cambio.

**🗑️ 8. deleteTask(id): Eliminar una Tarea**

****

* Llama a **/api/tasks/:id** con método **DELETE**.
* Si tiene éxito, recarga las tareas.

**🔚 9. logout(): Cerrar Sesión**

****

* Elimina el token y el nombre del **localStorage**.
* Oculta el dashboard.
* Muestra nuevamente la pantalla de login.

**🔐** Así el usuario cierra sesión de forma segura.

**✅ Conclusión**

Este archivo **script.js** es **el motor del frontend** de TaskFlow. Conecta todas las piezas:

* ✅ Interacción con el usuario (formularios, clics).
* ✅ Comunicación con el backend (fetch, API REST).
* ✅ Gestión del estado (localStorage, DOM).
* ✅ Actualización dinámica de la interfaz.

Es un ejemplo claro de cómo el **JavaScript moderno** permite crear aplicaciones web dinámicas, seguras y profesionales.

**📄 Explicación Detallada del Archivo *backend/database/connection.js***

Este archivo es el **puente entre tu aplicación Node.js y la base de datos MySQL**. Es esencial para que todo el sistema funcione, ya que permite que el backend guarde y recupere datos (usuarios, tareas) de forma permanente. A continuación, te explico cada línea de código de forma clara y detallada.

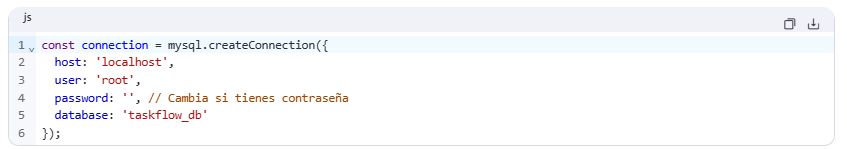
**🔧 1. Importar el Módulo MySQL**

****

* **require('mysql2'):** Importa la librería **mysql2**, que permite a Node.js comunicarse con una base de datos MySQL.
* **const mysql:** Guarda esa librería en una constante para usarla más adelante.

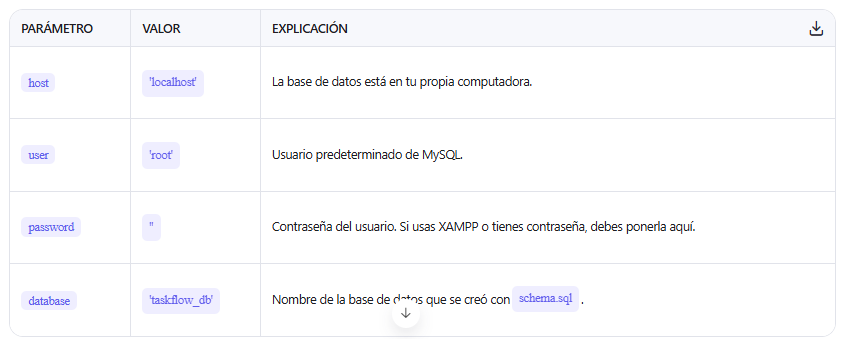
**✅ mysql2** es una versión mejorada de **mysql**, con soporte para **Promises** y mejor rendimiento.

**🔗 2. Crear la Conexión a la Base de Datos**

****

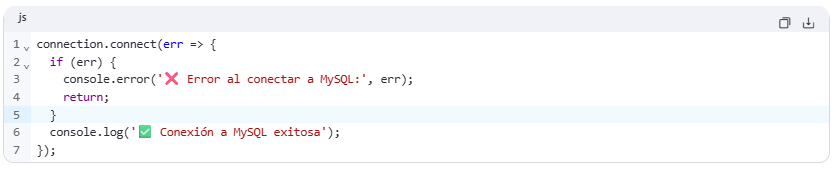
Aquí se configura cómo se conectará el backend a la base de datos.

**Parámetros de la Conexión:**

****

**⚠️ Importante:** Si usas XAMPP o tienes contraseña en MySQL, cambia **password: ''** por **password: 'tu\_contraseña'**.

**🌐 3. Establecer la Conexión**

****

* **connection.connect():** Intenta conectarse a la base de datos.
* **err:** Si hay un error (por ejemplo, MySQL no está encendido), este parámetro lo contiene.
* **Si hay error:**
  + Muestra un mensaje rojo en la consola: **❌ Error al conectar a MySQL: ...**
  + **return;** detiene la ejecución.
* **Si no hay error:**
  + Muestra un mensaje verde: **✅ Conexión a MySQL exitosa**

**✅** Este mensaje aparece cuando inicias el servidor con **node backend/server.js**.

**📦 4. Exportar la Conexión**

****

* **module.exports:** Permite que otros archivos del proyecto (como **task.js** o **user.js**) usen esta conexión.
* **connection:** Es el objeto que representa la conexión activa a MySQL.

**✅** Así, cualquier modelo que necesite hacer una consulta a la base de datos puede hacer:

****

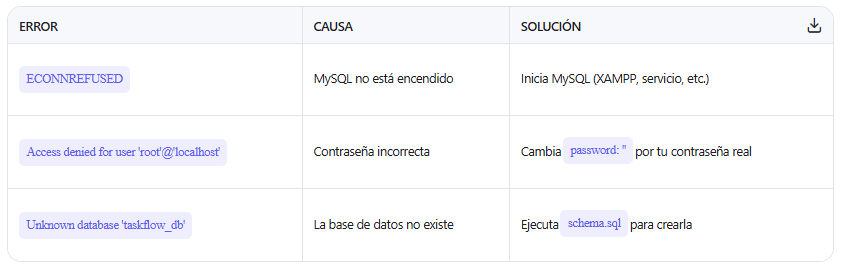
**Conclusión**

Este archivo **connection.js** es **el corazón de la persistencia de datos** en TaskFlow. Sin él, el backend no podría guardar usuarios ni tareas.

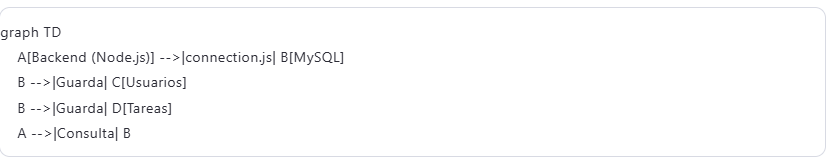
**¿Por qué es tan importante?**

* ✅ Permite que los datos **no se pierdan** al reiniciar el servidor.
* ✅ Es la **única forma** de que el backend hable con MySQL.
* ✅ Es reutilizable en todos los modelos (**user.js**, **task.js**).

**🛠️ Errores Comunes y Soluciones**

****

**🎯 Resumen Visual**

****

Este archivo es la **puerta de entrada** a tu base de datos. Sin él, TaskFlow no sería más que una interfaz sin memoria.

**📄 Explicación Detallada del Archivo *backend/models/task.js***

Este archivo es el **modelo de datos para las tareas** en tu aplicación **TaskFlow**. Es una de las piezas más importantes del **backend**, ya que define cómo se **crean, leen, actualizan y eliminan** las tareas en la base de datos MySQL. A continuación, te explico cada parte de forma clara y detallada.

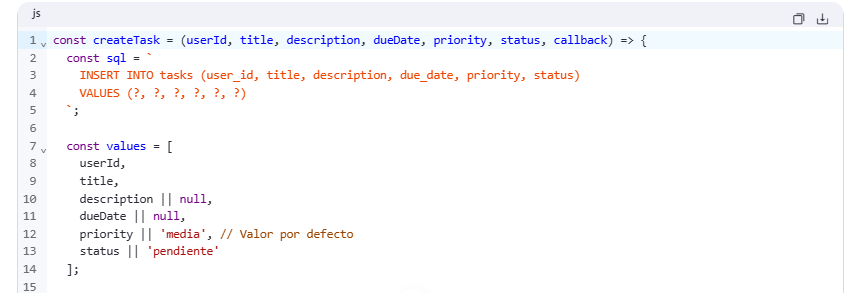
**🔗 1. Importar la Conexión a la Base de Datos**

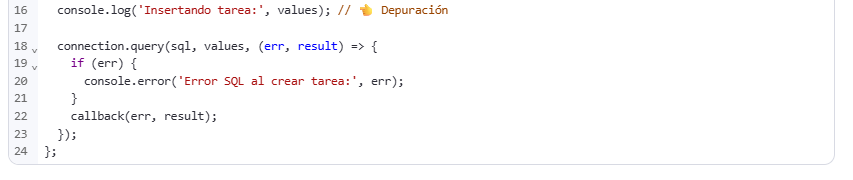
****

* **require('../database/connection'):** Importa el archivo que establece la conexión con MySQL.
* **connection:** Es el objeto que permite ejecutar consultas SQL (como INSERT, SELECT, UPDATE, DELETE).

**✅** Este es el mismo objeto que se crea en **connection.js** y que ya está conectado a **taskflow\_db**.

**✏️ 2. createTask(): Crear una Nueva Tarea**

****

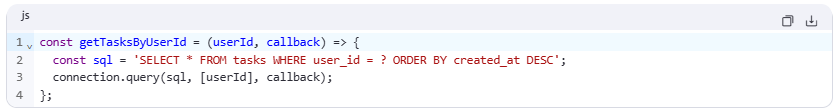
****

**Paso a Paso:**

1. **Parámetros de entrada:**
   * **userId:** El ID del usuario que crea la tarea.
   * **title:** Título de la tarea.
   * **description:** Descripción opcional.
   * **dueDate:** Fecha de vencimiento.
   * **priority:** Prioridad (baja, media, alta).
   * **status:** Estado (pendiente, en proceso, completada).
   * **callback:** Función que se ejecuta cuando termina la consulta.
2. **sql:** Comando SQL para insertar una nueva tarea.
   * Usa **?** como **placeholders** para evitar inyecciones SQL.
   * Inserta los datos en la tabla **tasks**.
3. **values:** Los valores que se insertarán en los **?**.
   * **description || null:** Si no hay descripción, guarda **null**.
   * **priority || 'media':** Si no se especifica, usa **'media'** como valor por defecto.
   * **status || 'pendiente':** Si no se especifica, usa **'pendiente'**.
4. **console.log():** Muestra en la consola qué datos se están insertando (útil para depurar).
5. **connection.query():** Ejecuta la consulta SQL.
   * Si hay error, lo muestra en rojo.
   * Llama al **callback** con el error o el resultado.

✅ Esta función es llamada desde **backend/routes/tasks.js** cuando el usuario crea una tarea.

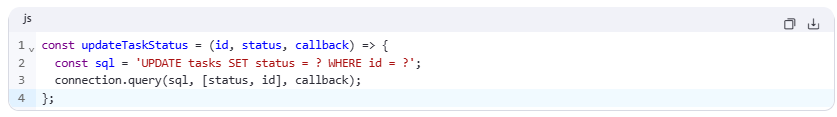
**📖 3. getTasksByUserId(): Obtener Tareas de un Usuario**

****

* **¿Qué hace?** Obtiene todas las tareas de un usuario específico.
* **SELECT \* FROM tasks:** Selecciona todos los campos de la tabla **tasks**.
* **WHERE user\_id = ?:** Filtra por el ID del usuario.
* **ORDER BY created\_at DESC:** Muestra las tareas más recientes primero.
* **[userId]:** El valor que reemplaza el **?**.
* **callback:** Recibe los resultados (las tareas) o un error.

**✅** Esta función es usada en **loadTasks()** del frontend para mostrar las tareas del usuario.

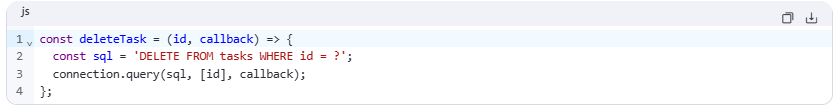
**🔄 4. updateTaskStatus(): Cambiar el Estado de una Tarea**

****

* **¿Qué hace?** Actualiza el campo **status** de una tarea específica.
* **UPDATE tasks SET status = ?:** Cambia el estado.
* **WHERE id = ?:** Solo afecta a la tarea con ese **id**.
* **[status, id]:** Valores que reemplazan los **?**.

**✅** Usado cuando el usuario cambia una tarea de "Pendiente" a "Completada".

**🗑️ 5. deleteTask(): Eliminar una Tarea**

****

* **¿Qué hace?** Elimina una tarea de la base de datos.
* **DELETE FROM tasks:** Borra un registro.
* **WHERE id = ?:** Solo borra la tarea con ese ID.
* **[id]:** El ID de la tarea a eliminar.

**✅** Usado cuando el usuario hace clic en el botón de eliminar (🗑️).

**📦 6. Exportar las Funciones**

****

* **module.exports:** Hace que estas funciones estén disponibles en otros archivos.
* Otros archivos (como **routes/tasks.js**) pueden hacer:

****

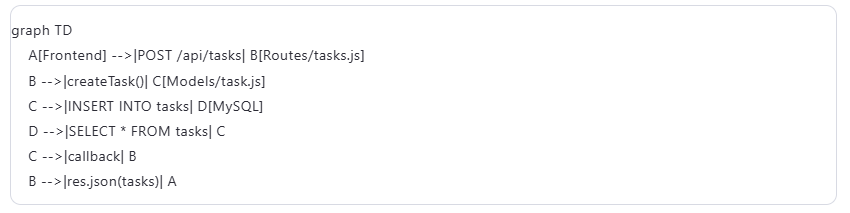
**Conclusión**

Este archivo **task.js** es el **modelo de negocio para las tareas**. Es una **capa de abstracción** entre el backend y la base de datos.

**¿Por qué es importante?**

* ✅ Centraliza toda la lógica de acceso a datos.
* ✅ Evita repetir código SQL en las rutas.
* ✅ Permite fácil mantenimiento: si cambias la tabla **tasks**, solo modificas este archivo.
* ✅ Es reutilizable y seguro (usa **?** para evitar inyecciones SQL).

**🎯 Resumen Visual**

****

Este modelo es esencial para que **TaskFlow** funcione como una **aplicación fullstack real y profesional**.

**📄 Explicación Detallada del Archivo *backend/models/user.js***

Este archivo es el **modelo de datos para los usuarios** en tu aplicación **TaskFlow**. Es una pieza fundamental del **backend**, ya que define cómo se **registran** y **buscan** los usuarios en la base de datos MySQL. A continuación, te explico cada parte de forma clara y detallada.

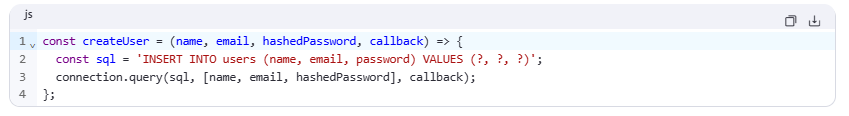
**🔗 1. Importar la Conexión a la Base de Datos**

****

* **require('../database/connection'):** Importa el archivo que establece la conexión con MySQL.
* **connection:** Es el objeto que permite ejecutar consultas SQL (como INSERT, SELECT) para interactuar con la base de datos **taskflow\_db**.

✅ Este es el mismo objeto que se crea en **connection.js** y que ya está conectado a la base de datos.

**✏️ 2. createUser(): Crear un Nuevo Usuario**

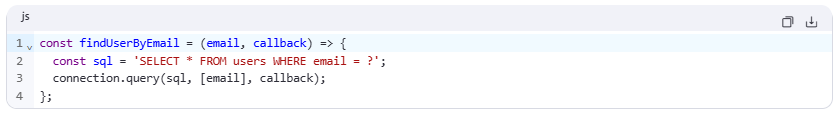
****

**Paso a Paso:**

1. **Parámetros de entrada:**
   * **name:** Nombre completo del usuario.
   * **email:** Correo electrónico (único).
   * **hashedPassword:** Contraseña **encriptada** con **bcrypt** (nunca se guarda en texto plano).
   * **callback:** Función que se ejecuta cuando termina la consulta.
2. **sql:** Comando SQL para insertar un nuevo usuario.
   * **INSERT INTO users:** Indica que se va a insertar un registro en la tabla **users**.
   * **(name, email, password):** Los campos de la tabla.
   * **VALUES (?, ?, ?):** Los valores que se insertarán. Los **?** son **placeholders** que previenen inyecciones SQL.
3. **[name, email, hashedPassword]:** Los valores que reemplazan los **?** en el orden correcto.
4. **connection.query():** Ejecuta la consulta SQL en la base de datos.
   * Si hay éxito, el usuario queda registrado.
   * El **callback** recibe el resultado o un error.

**✅** Esta función es llamada desde **backend/routes/auth.js** cuando un usuario se registra.

**🔍 3. findUserByEmail(): Buscar un Usuario por Correo**

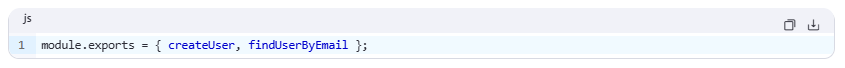
****

* **¿Qué hace?** Busca un usuario en la base de datos usando su correo electrónico.
* **SELECT \* FROM users:** Selecciona todos los campos del usuario (id, name, email, password, etc.).
* **WHERE email = ?:** Filtra por el correo electrónico.
* **[email]:** El valor que reemplaza el **?**.
* **callback:** Recibe el usuario encontrado (si existe) o **null** si no existe.

**✅** Esta función es crucial para el proceso de **login:**

1. El usuario ingresa su correo.
2. El sistema busca el usuario con **findUserByEmail()**.
3. Si lo encuentra, compara la contraseña ingresada con el **hashedPassword** almacenado.

**📦 4. Exportar las Funciones**

****

* **module.exports:** Hace que estas funciones estén disponibles en otros archivos del proyecto.
* Otros archivos (como **routes/auth.js**) pueden hacer:



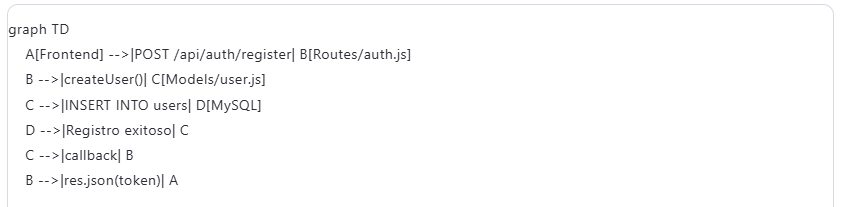
**Conclusión**

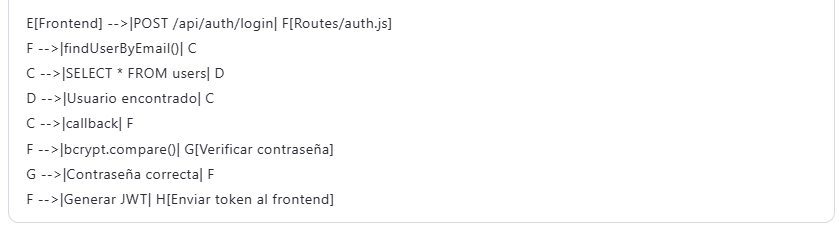
Este archivo **user.js** es el **modelo de negocio para los usuarios**. Es una **capa de abstracción** entre el backend y la base de datos.

**¿Por qué es importante?**

* ✅ Centraliza toda la lógica de acceso a datos de usuarios.
* ✅ Asegura que las contraseñas se manejen de forma segura (solo se almacena el hash).
* ✅ Permite fácil mantenimiento: si cambias la tabla **users**, solo modificas este archivo.
* ✅ Es reutilizable y seguro (usa **?** para evitar inyecciones SQL).

**🎯 Resumen Visual del Flujo de Registro e Inicio de Sesión**

****

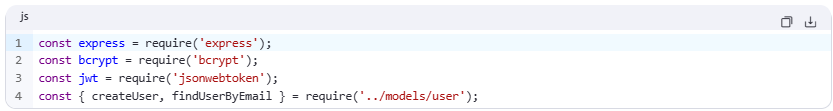
****

Este modelo es esencial para que **TaskFlow** tenga un sistema de autenticación **seguro y profesional**.

**📄 Explicación Detallada del Archivo *backend/routes/auth.js***

Este archivo es el **gestor de autenticación** de tu aplicación **TaskFlow**. Es una de las partes más críticas del backend, ya que controla cómo los usuarios **se registran** y **inician sesión** de forma segura. A continuación, te explico cada línea de código de forma clara y detallada.

**🔗 1. Importar Dependencias y Modelos**

****

* **express:** Framework para crear rutas en el servidor.
* **bcrypt:** Librería para **encriptar contraseñas**. Nunca se almacenan en texto plano.
* **jsonwebtoken (jwt):** Librería para generar **tokens de autenticación seguros** (JWT).
* **{ createUser, findUserByEmail }:** Funciones del modelo **user.js** para interactuar con la base de datos.

**✅** Estas librerías son esenciales para una autenticación segura.

**🛣️ 2. Crear el Router de Express**

****

* **express.Router():** Crea un manejador de rutas modular.
* Permite definir rutas como **/api/auth/register** y **/api/auth/login** en un archivo separado.

**🔐 3. Ruta POST /register: Registrar un Nuevo Usuario**

****

**Paso a Paso:**

1. **router.post('/register'):** Define una ruta que responde a solicitudes **POST** en **/api/auth/register**.
2. **const { name, email, password } = req.body;:**
   * Extrae los datos enviados desde el frontend (desde **script.js**).
   * Ejemplo: **{ "name": "Jhon", "email": "jhon@example.com", "password": "123456" }**.
3. **bcrypt.hash(password, 10):**
   * **Encripta la contraseña** con un "salt" de 10 rondas.
   * El resultado es un hash irreversible (ej: **$2b$10$abc123...**).
   * Esto es **crucial para la seguridad.**
4. **createUser(...):**
   * Llama a la función del modelo **user.js** para insertar el usuario en la base de datos.
   * Guarda: **name, email** y el **hashedPassword**.
5. **Manejo de errores y respuesta:**
   * Si hay un error (por ejemplo, correo duplicado), devuelve un error 400.
   * Si todo va bien, devuelve un código 201 ("Creado") y un mensaje.

**✅** Así se crea un usuario de forma segura.

**🔑 4. Ruta POST /login: Iniciar Sesión**

****

**Paso a Paso:**

1. **router.post('/login'):** Define una ruta para iniciar sesión.
2. **const { email, password } = req.body;:**
   * Obtiene el correo y contraseña enviados por el frontend.
3. **findUserByEmail(email, ...):**
   * Busca en la base de datos si existe un usuario con ese correo.
   * **results** es un array. Si está vacío (**length === 0**), el usuario no existe.
4. **Verificar la contraseña:**

****

* + **bcrypt.compare()** compara la contraseña ingresada con el hash almacenado.
  + Devuelve **true** si coinciden, **false** si no.

1. **Validación de credenciales:**
   * Si el usuario no existe o la contraseña es incorrecta, se devuelve un error 401 ("No autorizado").
2. **Generar el Token JWT:**

****

* + **jwt.sign():** Crea un token firmado.
  + **Payload: { id: user.id, name: user.name }** (información del usuario).
  + **Clave secreta: 'secret\_key'** (debe ser segura en producción).
  + **Caducidad:** El token dura 1 hora.

1. **Enviar respuesta al frontend:**

****

* + El frontend recibe el token y el nombre.
  + El token se guarda en **localStorage** y se usa para acceder a rutas protegidas.

**✅** Así el usuario inicia sesión de forma segura y obtiene acceso al dashboard.

**📦 5. Exportar el Router**

****

* **module.exports:** Hace que este conjunto de rutas esté disponible en otros archivos.
* Es usado en **backend/server.js** con:



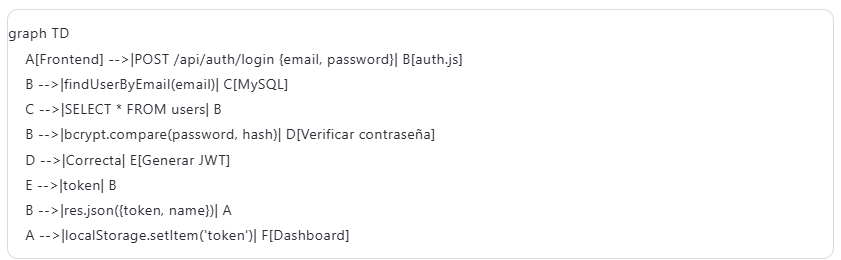
**Conclusión**

Este archivo **auth.js** es el **guardián de la seguridad** en TaskFlow. Maneja:

* ✅ **Registro seguro** con contraseñas encriptadas.
* ✅ **Inicio de sesión autenticado** con verificación de credenciales.
* ✅ **Tokens JWT** para mantener sesiones seguras.
* ✅ **Respuestas claras** al frontend (éxito o error).

Es un ejemplo perfecto de cómo implementar una autenticación robusta en una aplicación fullstack con Node.js.

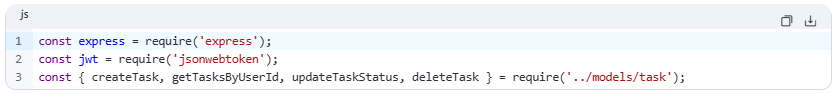
**🎯 Resumen Visual del Flujo de Login**

****

**📄 Explicación Detallada del Archivo *backend/routes/tasks.js***

Este archivo es el **gestor de rutas para las tareas** en tu aplicación **TaskFlow**. Es una de las partes más importantes del backend, ya que define cómo se **crean, leen, actualizan y eliminan** las tareas de forma segura y estructurada. A continuación, te explico cada parte de forma clara y detallada.

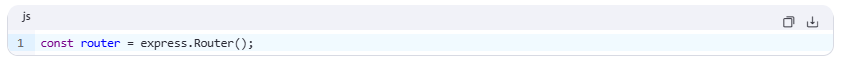
**🔗 1. Importar Dependencias y Modelos**

****

* **express:** Framework para crear rutas en el servidor.
* **jsonwebtoken (jwt):** Librería para verificar **tokens de autenticación** (JWT).
* **{ createTask, ... }:** Funciones del modelo **task.js** que interactúan con la base de datos.

**✅** Estas funciones son las que realmente modifican los datos en MySQL.

**🛣️ 2. Crear el Router de Express**

****

* **express.Router():** Crea un manejador de rutas modular.
* Permite definir rutas como **/api/tasks** en un archivo separado.

**🔐 3. Middleware authenticate: Proteger las Rutas**

****

**¿Qué hace?**

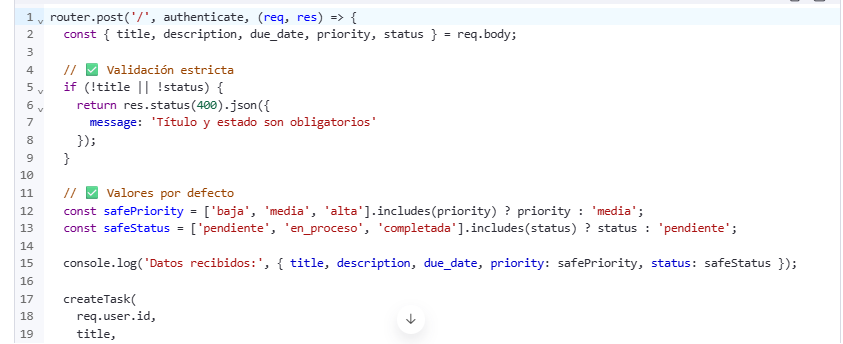
Este middleware **protege todas las rutas de tareas**. Solo usuarios autenticados pueden acceder a ellas.

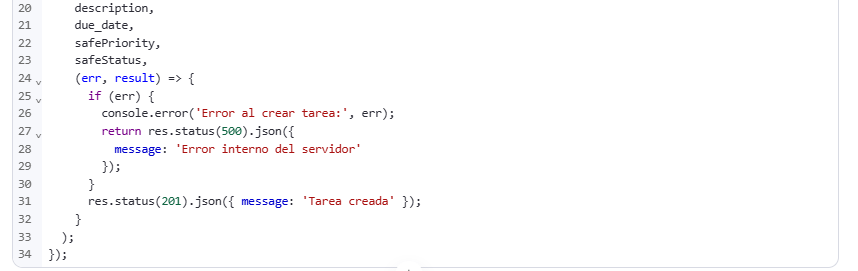
**Paso a Paso:**

1. **Obtener el token:**
   * **req.headers.authorization** contiene el encabezado **Authorization: Bearer <token>**.
   * **.split(' ')[1]** extrae solo el token (después de **Bearer**).
2. **Verificar si hay token:**
   * Si no hay token, responde con **401 Unauthorized**.
3. **Verificar el token con JWT:**
   * **jwt.verify(token, 'secret\_key', ...)** valida que el token sea correcto y no haya expirado.
   * Si es válido, extrae el **user** (con **id** y **name**).
   * Lo guarda en **req.user** para que las rutas lo usen.
4. **next():** Si todo está bien, permite que la solicitud continúe a la ruta.

**✅** Así se asegura que solo usuarios logueados puedan gestionar tareas.

**➕ 4. Ruta POST /api/tasks: Crear una Nueva Tarea**

****

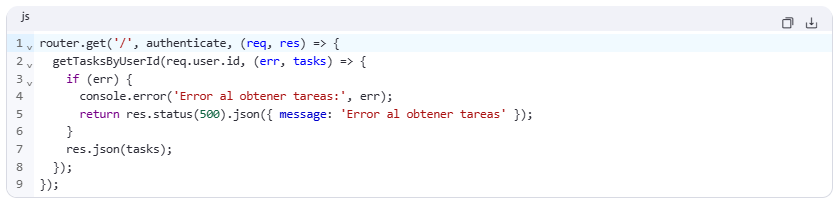
****

**Paso a Paso:**

1. **authenticate:** Primero verifica que el usuario esté logueado.
2. **Extraer datos del cuerpo (req.body):**
   * **title**, **description**, **due\_date**, **priority**, **status**.
3. **Validación:**
   * **title** y **status** son obligatorios.
   * Si faltan, devuelve un error 400.
4. **Valores seguros por defecto:**
   * Si **priority** no es **'baja'**, **'media'** o **'alta'**, usa **'media'**.
   * Si **status** no es válido, usa **'pendiente'**.
5. **Llamar a createTask():**
   * Pasa el **userId** (del token), los datos de la tarea y un **callback**.
6. **Respuesta:**
   * Si hay error (ej: base de datos caída), devuelve 500.
   * Si todo va bien, devuelve 201 ("Creado").

✅ Así se crea una tarea de forma segura y validada.

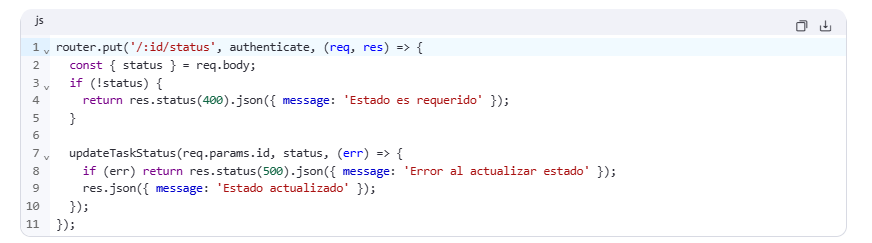
**📖 5. Ruta GET /api/tasks: Obtener Tareas del Usuario**

****

* **authenticate:** Solo usuarios logueados pueden ver sus tareas.
* **getTasksByUserId(req.user.id, ...):** Llama al modelo para obtener las tareas del usuario.
* Si hay error, devuelve 500.
* Si tiene éxito, devuelve las tareas en formato JSON.

**✅** Esta ruta es llamada por **loadTasks()** en el frontend.

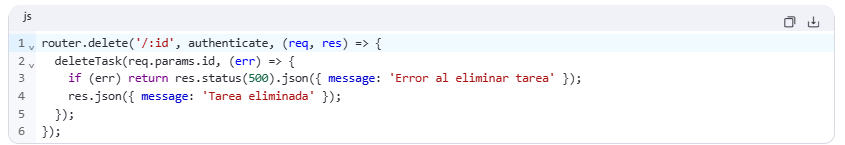
**🔄 6. Ruta PUT /api/tasks/:id/status: Cambiar Estado de una Tarea**

****

* **req.params.id:** El ID de la tarea (de la URL).
* **req.body.status:** El nuevo estado.
* **Validación:** El **status** es obligatorio.
* **updateTaskStatus():** Actualiza el estado en la base de datos.
* Responde con un mensaje de éxito o error.

**✅** Usado cuando el usuario cambia una tarea de "Pendiente" a "Completada".

**🗑️ 7. Ruta DELETE /api/tasks/:id: Eliminar una Tarea**

****

* **req.params.id:** ID de la tarea a eliminar.
* **deleteTask():** Borra la tarea de la base de datos.
* Responde con un mensaje de éxito o error.

**✅** Usado cuando el usuario hace clic en el botón de eliminar.

**📦 8. Exportar el Router**

****

* **module.exports:** Hace que este conjunto de rutas esté disponible en **backend/server.js**.
* Es usado con:



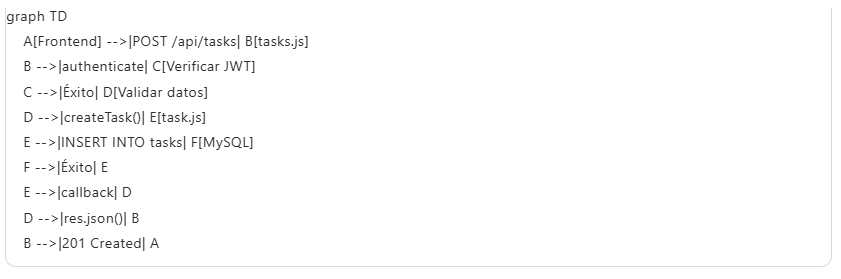
**Conclusión**

Este archivo tasks.js es el **controlador principal de las tareas**. Es una **capa de seguridad y validación** entre el frontend y el modelo de datos.

**¿Por qué es importante?**

* ✅ **Protege rutas** con autenticación JWT.
* ✅ **Valida datos** antes de guardarlos.
* ✅ **Maneja errores** de forma elegante.
* ✅ **Conecta el frontend con el modelo** (**task.js**).
* ✅ Es **modular y reutilizable**.

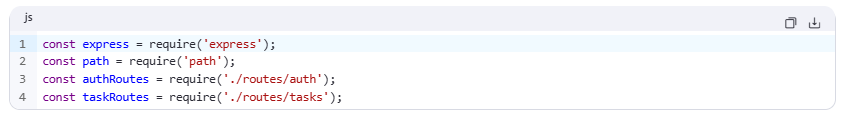
**🎯 Resumen Visual del Flujo de una Solicitud**

****

**📄 Explicación Detallada del Archivo *backend/server.js***

Este archivo es el **archivo principal del servidor** de tu aplicación **TaskFlow**. Es el punto de entrada del backend, el "corazón" que inicia el servidor, configura las rutas y sirve la aplicación al usuario. A continuación, te explico cada línea de código de forma clara y detallada.

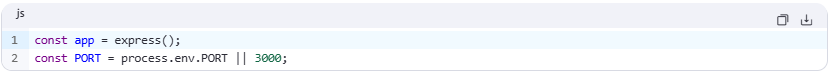
**🔗 1. Importar Dependencias y Rutas**

****

* **express:** Framework para construir servidores web en Node.js.
* **path:** Módulo de Node.js para manejar rutas de archivos y directorios de forma segura.
* **authRoutes:** Importa las rutas de autenticación(**/api/auth/register**, **/api/auth/login**).
* **taskRoutes:** Importa las rutas de tareas(**/api/tasks**).

**✅** Estas rutas fueron definidas en **routes/auth.js** y **routes/tasks.js**.

**🏗️ 2. Crear la Aplicación Express y Definir el Puerto**

****

* **app:** Es la instancia principal del servidor Express. Aquí se configuran todas las rutas, middlewares y comportamientos.
* **PORT:** Define el puerto del servidor.
  + **process.env.PORT:** Usa el puerto asignado por servicios en la nube (como Railway o Render).
  + **|| 3000:** Si no hay un puerto definido, usa el 3000 (para desarrollo local).

**✅** Así el servidor funciona tanto en local como en producción.

**⚙️ 3. Configurar Middlewares**

****

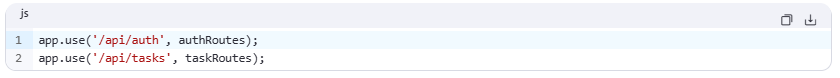
* **express.json():** Middleware que **parsea** el cuerpo de las solicitudes HTTP en formato JSON.
* Es esencial para poder usar **req.body** en las rutas (por ejemplo, **{ email, password }** en el login).

****

* **express.static():** Middleware que sirve archivos estáticos (HTML, CSS, JS, imágenes).
* **path.join(\_\_dirname, '../frontend'):**
  + **\_\_dirname:** Directorio actual (**backend/**).
  + **../frontend:** Sube un nivel y entra a la carpeta **frontend**.
* Esto permite que cuando accedas a **http://localhost:3000**, se sirva el **index.html** y todos sus recursos.

✅ Así el frontend se carga automáticamente.

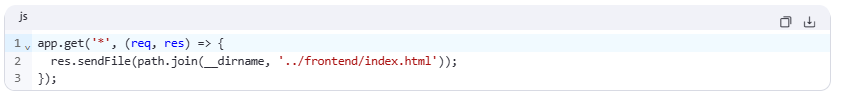
**🛣️ 4. Montar las Rutas del Backend**

****

* **app.use():** Monta un conjunto de rutas bajo un prefijo específico.
* **/api/auth:** Todas las rutas de **authRoutes** estarán bajo este prefijo.
  + Ej: **POST /api/auth/login**
* **/api/tasks:** Todas las rutas de **taskRoutes** estarán bajo este prefijo.
  + Ej: **GET /api/tasks**, **POST /api/tasks**

**✅** Esto organiza las rutas de forma limpia y modular.

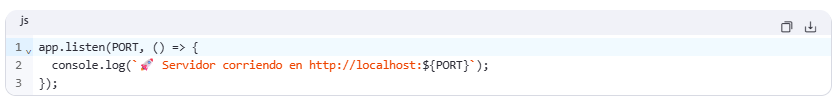
**🌐 5. Ruta de Respaldo (SPA - Aplicación de Página Única)**

****

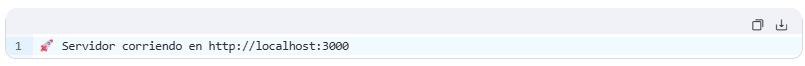
* **app.get('\*'):** Captura **todas las rutas GET** que no coincidan con las anteriores.
* **¿Por qué es necesaria?**
  + En una SPA (Single Page Application), el frontend maneja la navegación (login → dashboard).
  + Si el usuario recarga la página en **/dashboard**, el servidor debe seguir sirviendo **index.html**, no un error 404.
* **res.sendFile():** Envía el archivo **index.html** como respuesta.

✅ Así el frontend puede manejar la navegación con JavaScript sin problemas.

**🚀 6. Iniciar el Servidor**

****

* **app.listen():** Inicia el servidor y lo pone a escuchar en el puerto definido.
* Cuando el servidor se inicia, muestra un mensaje en la consola:



Este es el mensaje que ves cuando ejecutas **node backend/server.js**.

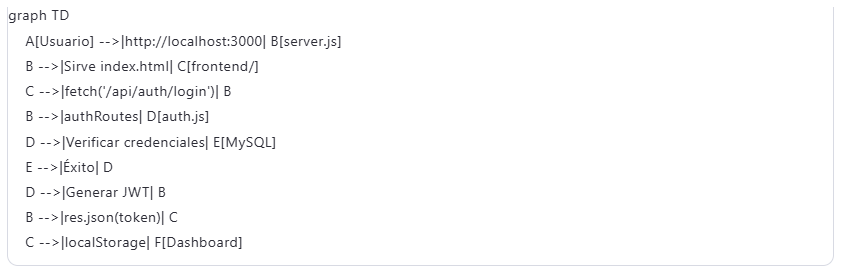
**✅ Conclusión**

Este archivo **server.js** es el **punto de entrada y control del backend**. Es como el **director de orquesta** que coordina todas las partes:

* ✅ Inicia el servidor con Express.
* ✅ Configura el manejo de JSON y archivos estáticos.
* ✅ Conecta las rutas de autenticación y tareas.
* ✅ Sirve la aplicación frontend.
* ✅ Permite que la app funcione como una SPA.
* ✅ Funciona tanto en local como en producción.

Sin este archivo, **nada de TaskFlow funcionaría**.

**🎯 Resumen Visual del Flujo del Servidor**

****

**📄 Explicación Detallada del Archivo *database/schema.sql***

Este archivo es el **esquema de la base de datos** de tu aplicación **TaskFlow**. Define cómo se estructuran los datos en MySQL, creando las tablas y relaciones necesarias para que el sistema funcione. A continuación, te explico cada línea de código de forma clara y detallada.

**🛠️ 1. Crear la Base de Datos**

****

* **CREATE DATABASE:** Comando SQL para crear una nueva base de datos.
* **IF NOT EXISTS:** Verifica si ya existe una base de datos llamada **taskflow\_db**. Si existe, no la crea de nuevo (evita errores).
* **taskflow\_db:** Nombre de la base de datos donde se almacenarán todos los datos del proyecto.

**✅** Esta línea asegura que la base de datos exista antes de crear las tablas.

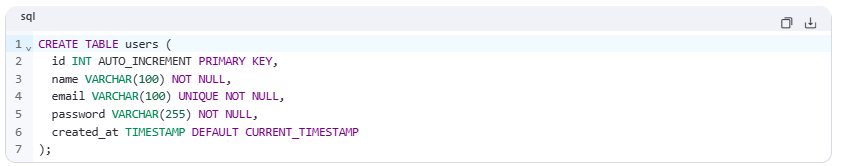
**🔄 2. Usar la Base de Datos**

****

* **USE:** Indica al servidor MySQL que todas las siguientes operaciones (como crear tablas) se realizarán dentro de la base de datos **taskflow\_db**.
* Es como "entrar" a una carpeta específica antes de crear archivos.

**✅** Sin esta línea, las tablas podrían crearse en otra base de datos por defecto.

**🧑 3. Crear la Tabla users**

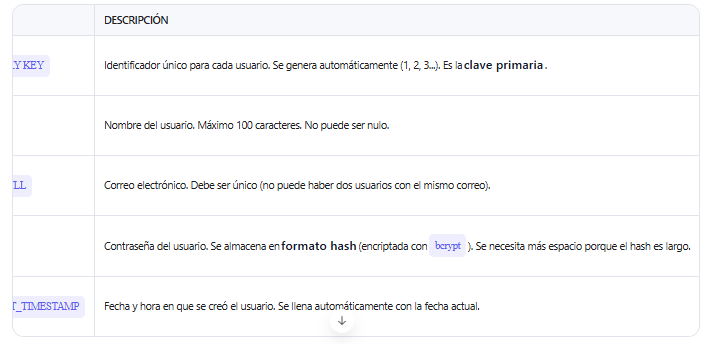
****

**¿Qué es esta tabla?**

Almacena la información de los usuarios registrados en TaskFlow.

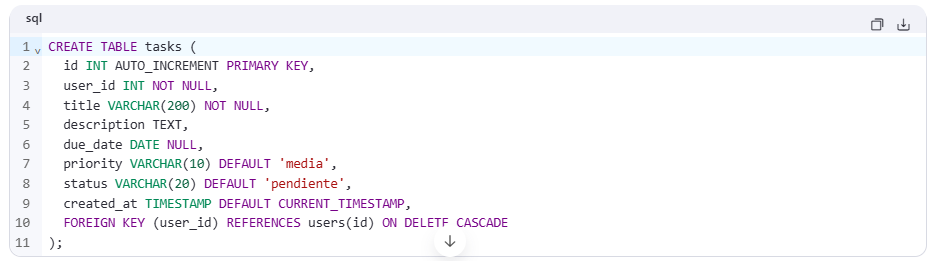
**Descripción de cada campo:**

****

****

**✅** Esta tabla es esencial para el sistema de autenticación.

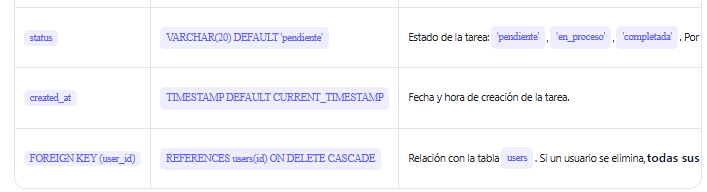
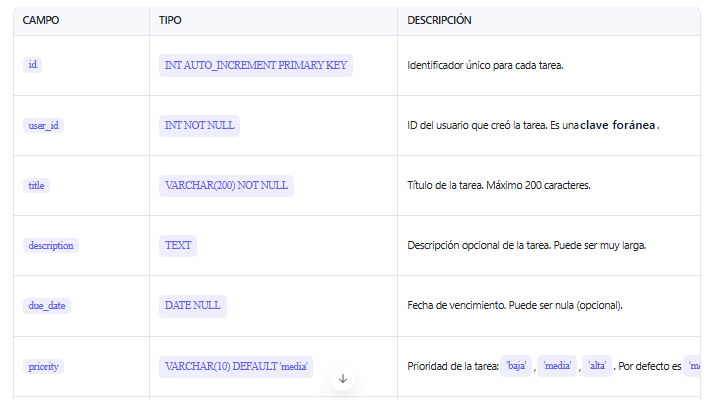
**📝 4. Crear la Tabla tasks**

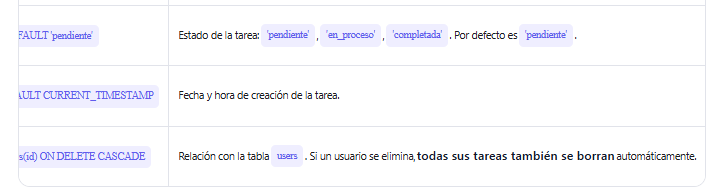
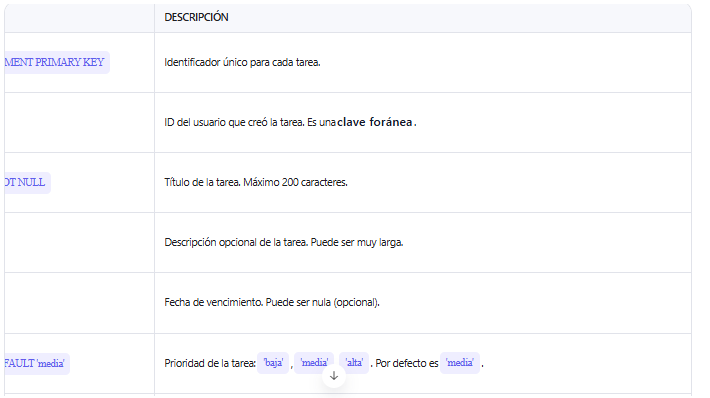
****

**¿Qué es esta tabla?**

Almacena todas las tareas creadas por los usuarios.

**Descripción de cada campo:**

****

****

✅ Esta tabla permite que cada usuario tenga sus propias tareas, organizadas por prioridad y estado.

**🔗 5. Relación entre Tablas (Clave Foránea)**

****

* **FOREIGN KEY:** Crea una relación entre **tasks.user\_id** y **users.id**.
* **REFERENCES users(id):** Indica que **user\_id** debe coincidir con un **id** válido en la tabla **users**.
* **ON DELETE CASCADE:** Si un usuario se elimina, **todas sus tareas se eliminan automáticamente**.Esto mantiene la integridad de los datos.

**✅** Evita tareas huérfanas (sin usuario asociado).

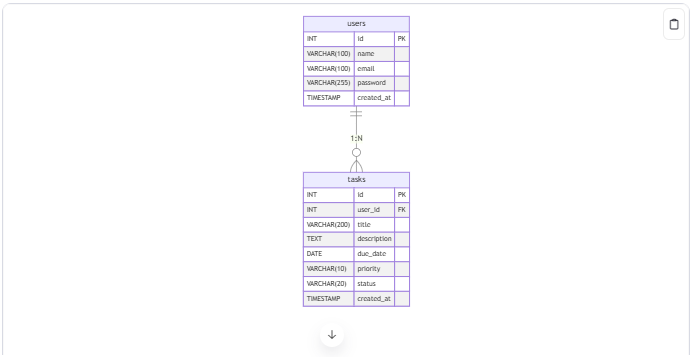
**✅ Conclusión**

Este archivo **schema.sql** es el **plan maestro de la base de datos.** Define:

* **✅** La estructura de las tablas(**users** y **tasks**).
* ✅ Los tipos de datos y restricciones (NOT NULL, UNIQUE, PRIMARY KEY).
* ✅ La relación entre usuarios y tareas.
* **✅** Valores por defecto para campos como **priority** y **status**.

Es fundamental para que el backend pueda guardar y recuperar datos de forma segura y organizada.

**🎯 Resumen Visual de la Base de Datos**

****

Este esquema garantiza que **TaskFlow** sea unaaplicación **fullstack completa, segura y funcional**.

**📄 Explicación del Archivo *.gitignore***

El archivo que mencionas (aunque no es un archivo de código como los anteriores) es un **.gitignore**, un archivo de configuración esencial en cualquier proyecto de desarrollo. Aunque no contiene código ejecutable, su función es **muy importante** para mantener tu repositorio limpio y seguro.

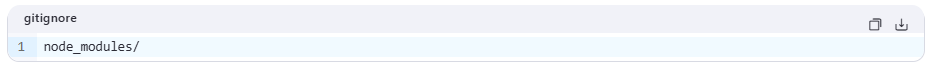
**📁 ¿Qué es .gitignore?**

El archivo **.gitignore** le dice a **Git** (el sistema de control de versiones) **qué archivos o carpetas NO debe subir a GitHub**.

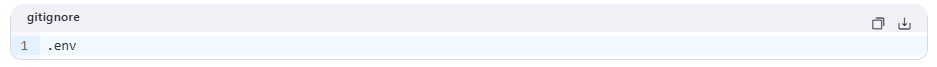
Esto es crucial porque:

* Evita subir archivos sensibles (como contraseñas).
* Reduce el tamaño del repositorio.
* Evita subir archivos temporales o generados automáticamente.

**🔍 Explicación Detallada de Cada Línea**

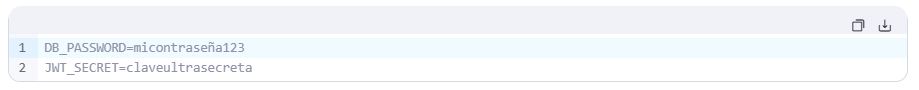
****

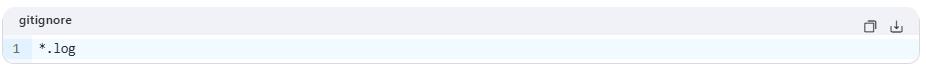
* **¿Qué es?** La carpeta **node\_modules** contiene todas las dependencias de Node.js (como **express**, **bcrypt**,etc.)instaladas con **npm install**.
* **¿Por qué ignorarla?**
  + Es muy grande (puede tener miles de archivos).
  + No es necesaria en GitHub porque cualquier persona puede recrearla con **npm install**.
  + Se genera automáticamente.



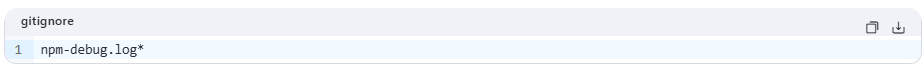
* **¿Qué es?** Un archivo **.env** almacena **variables de entorno**, como contraseñas de base de datos, claves secretas de JWT o API keys.
* **¿Por qué ignorarla?**
  + Es **altamente sensible**. Si lo subes a GitHub, cualquiera puede ver tus contraseñas.
  + Debes crear un **.env.example** con claves ficticias para que otros sepan qué variables necesitan.

✅ Ejemplo de **.env**:

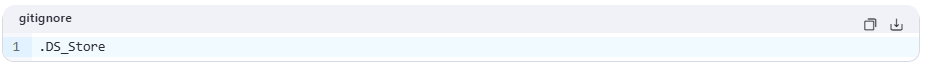




* **¿Qué son?** Archivos de registro (logs) generados por el sistema o el servidor.
* **¿Por qué ignorarlos?**
  + Son archivos temporales.
  + Pueden contener información sensible.
  + Cambian constantemente y no aportan valor al código.



* **¿Qué es?** Archivos de depuración generados por **npm** cuando hay errores.
* **¿Por qué ignorarlos?**
  + Son temporales.
  + No son parte del código fuente.
  + Pueden ser grandes y ruidosos.



* **¿Qué es?** Un archivo oculto que crea **macOS** para almacenar configuraciones visuales de carpetas (iconos, posición, etc.).
* **¿Por qué ignorarla?**
  + Es específico del sistema operativo.
  + No es relevante para el proyecto.
  + No existe en Windows o Linux.

⚠️ Si usas Windows, también podrías ignorar **Thumbs.db**.

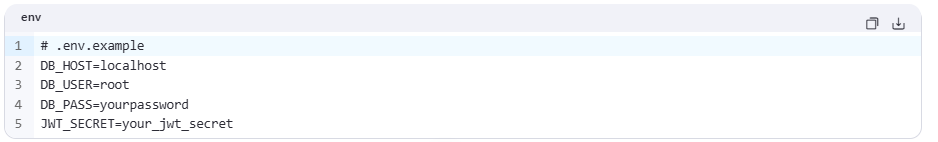
**✅ Conclusión**

Este archivo **.gitignore** es **esencial para un proyecto profesional** porque:

* ✅ **Protege tu seguridad** (no subes contraseñas).
* ✅ **Mantiene el repositorio limpio** (no subes archivos innecesarios).
* ✅ **Facilita la colaboración** (otros pueden clonar y usar npm install).
* ✅ **Sigue buenas prácticas** de desarrollo.

**🎯 ¿Qué deberías hacer?**

1. **Crea este archivo** en la raíz de tu proyecto (**taskflow/.gitignore**).
2. **Asegúrate de que esté en GitHub** (pero que no suba **node\_modules/** ni **.env**).
3. **Crea un** **.env.example** para que otros sepan qué variables necesitan.

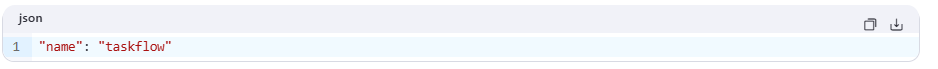


Este archivo, aunque pequeño, es una **pieza clave de la profesionalidad y seguridad** de tu proyecto TaskFlow.

**📄 Explicación Detallada del Archivo *package.json***

Este archivo es el **archivo de configuración principal** de tu proyecto **TaskFlow**. Es esencial para cualquier proyecto de Node.js, ya que define las **dependencias, scripts, metadatos y configuración del entorno**. A continuación, te explico cada parte de forma clara y detallada.

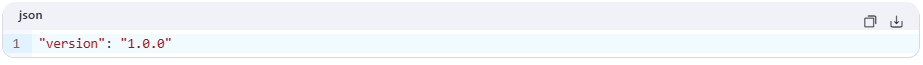
**📦 1. name: Nombre del Proyecto**

****

* Define el nombre de tu proyecto.
* Debe ser único si planeas publicarlo en npm, pero para proyectos locales es solo un identificador.
* Se usa en mensajes, documentación y herramientas.

✅ Este es el nombre que aparecerá en tu repositorio de GitHub.

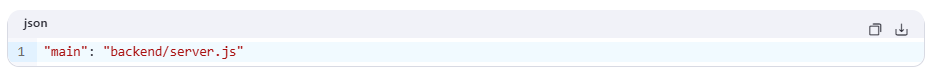
**📅 2. version: Versión del Proyecto**

****

* Sigue el sistema de **versionado semántico (SemVer)**: **MAJOR.MINOR.PATCH**
  + **1.0.0:** Versión inicial estable.
  + **1.0.1:** Corrección de errores.
  + **1.1.0:** Nueva funcionalidad.
  + **2.0.0:** Cambios importantes.

**✅** Es útil para gestionar actualizaciones y compatibilidad.

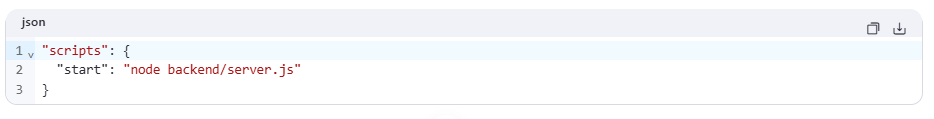
**🏁 3. main: Punto de Entrada del Proyecto**

****

* Indica cuál archivo se ejecuta cuando alguien **requiere tu proyecto** como un módulo.
* En este caso, si alguien hiciera **require('taskflow')**,se cargaría **backend/server.js**.
* También es el archivo que se ejecuta con **node .** o **npm start**.

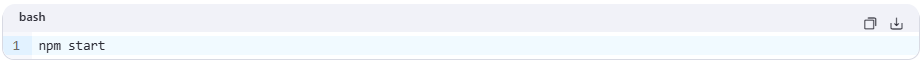
**✅** Este es el "corazón" del servidor.

**▶️ 4. scripts: Comandos Personalizados**

****

* Define comandos que puedes ejecutar con **npm run <nombre>**.
* El script más importante es **start**.

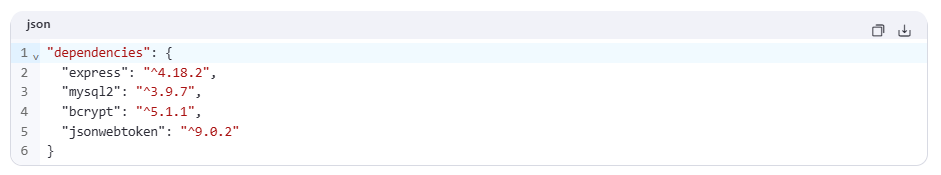
**¿Cómo se usa?**

****

* Este comando ejecuta **node backend/server.js**.
* Es la forma estándar de iniciar un servidor Node.js.

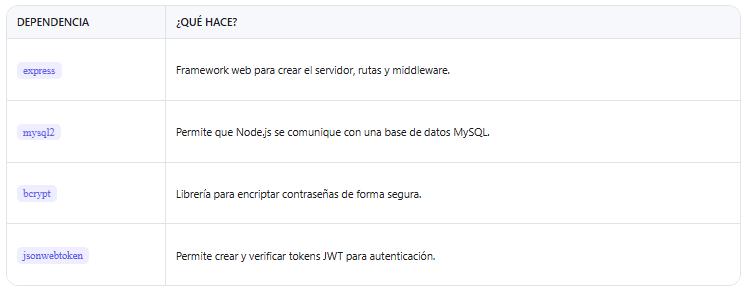
✅ Es esencial para desplegar en servicios como Railway o Render, que usan **npm start** para correr tu app.

**🔗 5. dependencies: Dependencias del Proyecto**

****

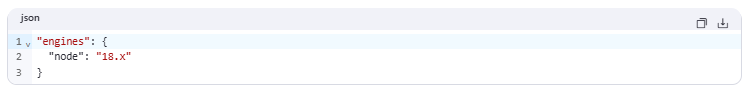
Son las **librerías externas** que tu proyecto necesita para funcionar. Se instalan con **npm install**.

**Descripción de cada dependencia:**

****

**🔐** El **^** antes de la versión(ej: **^4.18.2**)significa que npm puede instalar versiones compatibles (como **4.18.3** o **4.19.0**), pero no **5.0.0**.

**⚙️ 6. engines: Versión de Node.js Requerida**

****

* **Recomienda** o **requiere** una versión específica de Node.js.
* **"node": "18.x"** indica que el proyecto está diseñado para funcionar con **Node.js versión 18** (cualquier subversión como 18.17.0).

**¿Por qué es importante?**

* Algunas librerías o funciones solo funcionan en ciertas versiones.
* Servicios como **Railway** o **Render** usan esta información para configurar el entorno correctamente.

**✅** Ayuda a evitar errores de compatibilidad.

**✅ Conclusión**

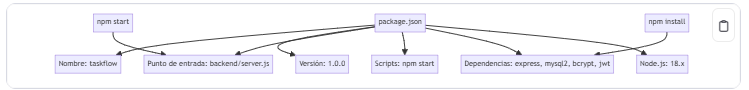
Este archivo **package.json** es el "**manual de instrucciones**" de tu proyecto. Define:

* **✅** Qué es el proyecto (**name**, **version**).
* **✅** Cómo se inicia (**scripts.start**).
* **✅** Qué librerías necesita (**dependencies**).
* **✅** Dónde está el punto de entrada (**main**).
* **✅** Qué versión de Node.js usar (**engines**).

Es fundamental para que:

* Tú puedas desarrollar.
* Otros puedan clonar y ejecutar tu proyecto.
* Servicios en la nube puedan desplegarlo automáticamente.

**🎯 Resumen Visual**

****