



**JUAN JOSE DELUQUEZ HERNANDEZ**

**UNIVERSIDAD DE TECMILENIO**

**FULLSTACK**

**ACTIVIDAD 2**

**21/01/2026**

## Reporte de Actividad 2: Gestor de Tareas con JavaScript

### Introducción

El objetivo de esta actividad fue desarrollar una aplicación interactiva para la gestión de tareas. El desarrollo se centró en la lógica funcional de JavaScript puro, aplicando conceptos de manipulación del DOM, sintaxis moderna ES6+ y Programación Orientada a Objetos (POO).

### 1. Aplicación de JavaScript Básico y Manipulación del DOM

Para la interacción con la interfaz de usuario definida en HTML, se utilizaron selectores básicos y eventos:

- Selección de Elementos: Se empleó `document.getElementById()` para obtener referencias a los elementos clave como el input de texto (`input-tarea`), el botón de agregar (`btn-agregar`) y la lista desordenada (`lista-tareas`).
- Eventos: Se utilizó `addEventListener('click')` en el botón para detonar la función de agregar tarea.
- Renderizado Dinámico: La lista de tareas no está escrita en el HTML estático. En su lugar, se inyecta dinámicamente modificando la propiedad `.innerHTML` del elemento `<ul>` cada vez que hay un cambio en el arreglo de tareas.

### 2. Implementación de Características ES6+

Se modernizó la sintaxis del código utilizando estándares actuales de JavaScript:

- Variables (`let` y `const`): Se utiliza `const` para referencias que no cambian (como las instancias de clases o selectores fijos) y `let` para variables dentro de bucles o que requieren reasignación.
- Template Literals: Para generar el HTML de cada tarea, se utilizaron las comillas invertidas. Esto permitió insertar variables directamente en la cadena de texto (ej. `${tarea.nombre}`) sin necesidad de concatenaciones complejas con el signo `+`.
- Arrow Functions: Se utilizaron funciones flecha `() => {}` en los callbacks de los eventos y en métodos de arreglo como `.map()` y `.filter()` para una sintaxis más limpia y concisa.

### 3. Programación Orientada a Objetos (POO)

La lógica de negocio se estructuró mediante clases para encapsular la funcionalidad, siguiendo el principio de responsabilidad única:

- Clase Tarea: Actúa como el modelo de datos. Su constructor inicializa el nombre, el estado (completada/pendiente) y genera un ID único basado en la fecha (`Date.now()`). Incluye métodos propios como `toggleEstado()` y `editarNombre()`, lo que permite que cada objeto gestione sus propios datos.
- Clase GestorDeTareas: Actúa como el controlador principal.
  - Mantiene el arreglo de tareas (`this.tareas`).
  - Contiene la lógica para Agregar (instanciando `new Tarea`), Eliminar (usando `.filter`), Editar y Cambiar Estado (buscando por ID).
  - Centraliza la actualización de la interfaz mediante un método `render()`, asegurando que la vista siempre coincida con los datos.

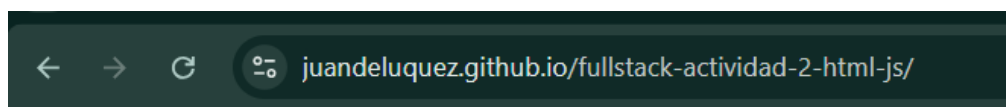
### 4. Persistencia de Datos (LocalStorage)

Para cumplir con el desafío adicional, se implementó persistencia de datos:

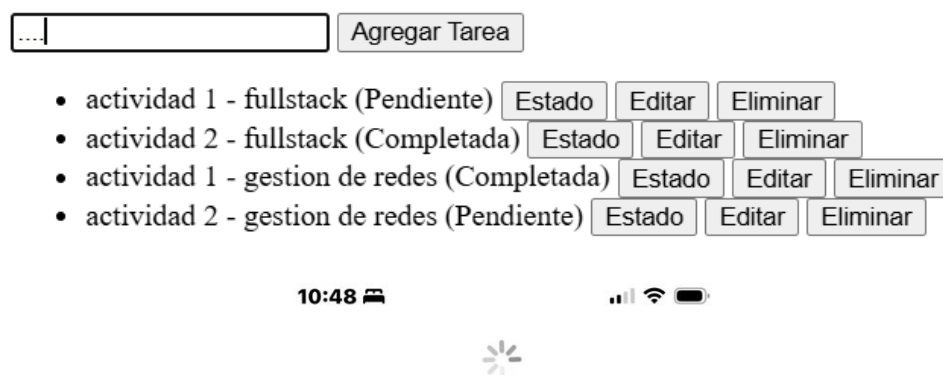
- Guardado: Cada vez que el gestor actualiza la lista (agrega, borra o edita), se invoca `localStorage.setItem`, convirtiendo el arreglo de objetos a una cadena de texto con `JSON.stringify()`.
- Recuperación: Al instanciar el `GestorDeTareas`, se intenta leer del `localStorage`. Si existen datos, se utiliza `JSON.parse()` y, crucialmente, se reconstruyen las instancias de la clase `Tarea` usando `.map()`. Esto es necesario porque `JSON.parse` devuelve objetos genéricos que no tienen los métodos de la clase (como `toggleEstado`).

## 5. Capturas de Pantalla

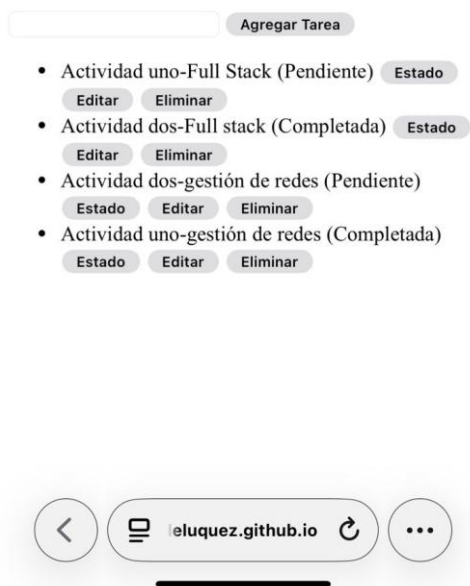
Captura 1: Vista inicial y agregar tarea



# Gestor de Tareas



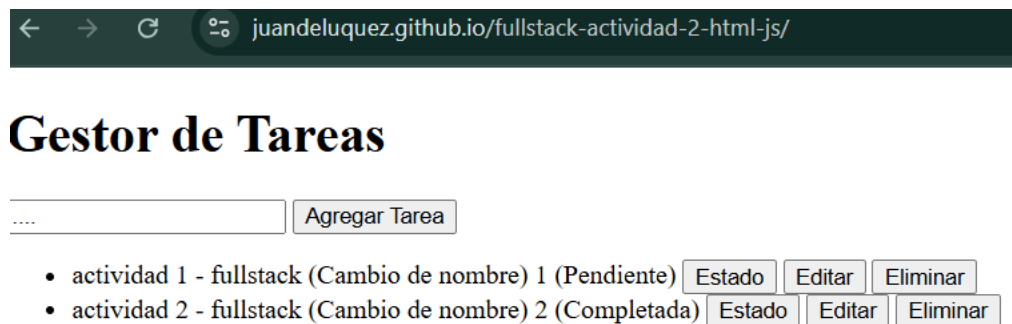
# Gestor de Tareas



Captura 2: Tarea editada y cambio de estado



Captura 3: Eliminación de tarea y persistencia



## Conclusión

La actividad permitió reforzar la importancia de separar la lógica de los datos de la interfaz visual. El uso de POO facilitó la escalabilidad del código, haciendo sencillo agregar nuevas funcionalidades como la persistencia.

[Repositorio](#)

[Gestor de Tareas](#)