Nombre del maestro: Tejeda Chávez Efraín Ulises.
Nombre del alumno: Ramírez Arreola Daniel Alejandro.

Práctica de JavaScript

Operaciones con Usuarios, Tareas y Etiquetas

Objetivo: Crear un conjunto de funciones en JavaScript que permitan simular operaciones de
altas, bajas, cambios y consultas en una app de gestión de tareas.

Descripción de la aplicación

Se desea tener un conjunto de funciones en JavaScript que permitan simular operaciones de
altas, bajas, cambios y consultas para la parte de Usuarios, Tareas y Etiquetas.
Esta práctica contiene los siguientes puntos:

1. Consideraciones
2. Configuración inicial
3. Creación de modelos
4. Funciones de Usuarios
5. Funciones del CRUD de Tareas y Etiquetas
6. Pruebas y ejemplos
7. Evaluación

Consideraciones

Considerar los siguientes aspectos para la implementación de la práctica:

1. Utilizaremos los archivos .html que tenías en la práctica 1. En una carpeta “P02”, crea
una carpeta “views”, una carpeta “models” y una carpeta “controllers”. Dentro de views,
pon tus archivos P01_home.html, P01_login.html y P01_tasks.html, pero con el nombre
home.html, login.html y tasks.html. Aunque para esta práctica en particular no necesitaremos los HTML (no hace falta que pongas las imágenes ni las importes
correctamente), te recomiendo que ya tengas la estructura armada.
2. Mostraremos la funcionalidad de nuestro conjunto de funciones desde la consola del
navegador.

3. En esta práctica no se espera aun la interacción con la página web (solo mostrar información en consola). No se habilitarán botones, ni modales, no se actualizará la interfaz, etc.

Configuración inicial

Crea los siguientes archivos de JavaScript dentro de tu carpeta “controllers”.

- users_controller.js
- tasks_controller.js
- tags_controller.js

Además, dentro de tu carpeta “models”, crea los siguientes archivos:

- user.js
- task.js
- tag.js

Y por último, crea un archivo “test.js” en la raíz de tu carpeta “P02”, donde haremos las pruebas y tendremos nuestro objeto de información, de manera que tengas una estructura de archivos como la siguiente:

En tu archivo home.html que tienes en “views”, carga los scripts de JavaScript en cualquier lugar (en head o en body). OJO, el orden es importante. Recuerda que si requieres una función de “Archivo A” en “Archivo B”, debes cargar primero “Archivo A”.

IMPORTANTE: NO NECESITAS utilizar nada de “import” ni “export” en ningún archivo.

Creación de Modelos

En el contexto de lo que estamos programando, un Modelo es una representación estructurada de un Objeto, que ya tiene ciertos atributos y reglas predefinidos para su manipulación. Como ya te puedes imaginar, para esta parte utilizaremos los archivos de la carpeta “models” (tag.js, task.js y user.js). Nota que los nombres están en singular, ya que un Modelo representa un único objeto, pero se utilizará para poder crear múltiples objetos basados en ese modelo. Es como un cascarón.

Los modelos deben de estar cada uno en su archivo correspondiente, y cada archivo debe de contener 3 cosas: una Función Generadora para obtener un ID (llamada getNextUserID, getNextTaskID y getNextTagId dependiendo del modelo), una Excepción y una

Clase.
Con esto en cuenta, cada uno de los archivos de los Modelos deberían de quedarte algo así:
Sí, puedes ver la cantidad de líneas de código que yo utilicé, pero es sólo una referencia. Tus archivos pueden tener más o menos líneas de código de acuerdo con cómo prefieras programarlo. Revisemos cada una de las 3 cosas que contiene el modelo:

- Función getNextID: Fíjate que cada función tiene el nombre correspondiente al modelo en el que está. Cuando comencemos a trabajar en los controladores, esta función deberá de regresarte el siguiente consecutivo de los elementos que tengas de esa clase (User, Task, Tag). Ya llegaremos a eso. Por lo pronto, has que únicamente regrese un 1.
- UserException / TaskException / TagException: Una excepción es un mecanismo que ocurre durante la ejecución de un programa para evitar errores. En este caso, cada uno de los modelos tiene su propia excepción, aunque las tres excepciones hacen exactamente lo mismo: mostrar un mensaje de error. Dado que las estamos declarando como clases, deben de tener su constructor, y únicamente su mensaje de error:
- Clase User / Task / Tag:

Dentro de cada una de las clases, debes de tener definidos los atributos (enlistados más adelante), el constructor de la clase y los getters y setters de cada atributo.
IMPORTANTE:
Cada atributo debe de tener su getter y setter, ya que en esta ocasión TRABAJARÁS CON ATRIBUTOS PRIVADOS (sólo accesibles desde dentro del objeto). Investiga por tu cuenta lo que esto implica y la sintáxis correcta de su declaración. Para cumplir con esto, deberás de tener algo parecido a lo siguiente dentro de la declaración de la clase:
ATENTO AL EJEMPLO: Fíjate cómo está la declaración de cada uno de los atributos.
Entiende cómo funciona la manera de manejar los datos privados (con el #) y cómo los asignas en el constructor. Fíjate cómo el setter lanza la excepción bajo la regla de que los ID se generan automáticamente con la función getNextTagID (más sobre las

reglas más
adelante).
Dentro de cada uno de los modelos, tendremos las clases correspondientes a dicho modelo
con los atributos y las reglas para la manipulación de la información. A continuación describo
la información que se requiere en cada uno de los modelos:
Modelo Atributos Reglas
User id
name
email
password
joined_at
• id: autogenerado a partir de una función getNextUserID(). NO debe poderse modificar una vez generado.
• name: No puede estar vacío.
• email: No puede estar vacío. No se puede repetir entre usuarios.
• password: No puede estar vacío. Debe tener al menos 8 caracteres.
• joined_at: Autogenerado con la fecha actual. NO debe poderse modificar una vez generado.
Task id
title
description
due_date
owner
status
tags
• id: autogenerado a partir de una función getNextTaskID();. NO debe poderse modificar una vez generado.
• title: No puede estar vacío.
• due_date: Debe ser un formato que new Date() acepte (que no regrese “Invalid Date”.
• owner: Debe ser un id que exista entre tus usuarios.
• status: Debe tener uno de los siguientes valores: A/F/C (Active, Finished, Cancelled).
• tags: Debe ser un arreglo (puede estar vacío).
Tag id
name
color
• id: autogenerado a partir de una función getNextUserID();. NO debe poderse modificar una vez

generado.
• name: No puede estar vacío.
• color: No puede estar vacío. Debe ser un valor en
hexadecimal.
Consideraciones importantes sobre los modelos:
• Los campos que NO tengan reglas son libres (por ejemplo, “description” en el
modelo
de Task).
• Debes de hacer las validaciones adecuadas para que se cumplan las reglas.
Dichas
validaciones las puedes hacer dentro del setter del atributo correspondiente, de
manera que si alguna validación no se cumple, lances una excepción indicándo
que no
se cumple.
• La fecha actual se genera cuando utilizas new Date() sin parámetros.
• Para verificar que el email NO se repita entre usuarios (email en modelo de
User) y que
el usuario sí exista para ser un owner (campo owner en modelo de Task),
debes de
utilizar find o findByIndex en tu objeto de información. Dicho objeto se explica
más
adelante. Por lo pronto, puedes omitir esta validación.
• Investiga cómo validar que el formato de color sea correcto (debe ser un
string con
este formato: “#ff00ff”).
Una vez que hayas armado toda esta estructura, debes de poder utilizar tus
clases de User,
Task y Tag correctamente. Antes de seguir avanzando en la práctica...

PRUÉBALO.

¿Cómo lo pruebas?
Asegúrate de importar tus modelos en el archivo home.html correctamente. Si
lo hiciste bien,
debes de poder abrir tu archivo home.html desde live server para probar que
las clases
funcionan correctamente desde la consola de tu navegador cuando las llamas
para crear
nuevos elementos:
También debes poder validar tus excepciones:
Si todo funciona bien para tus 3 modelos (validalo con todas las excepciones,
creando varios
elementos), entonces ahora podemos avanzar a los controladores.

Funciones de Usuarios

Ahora que ya tenemos el Modelo correcto para nuestra información, comenzaremos a crear registros nuevos en un objeto de información para “simular” registros en una base de datos.

Ese objeto de información será donde tendremos nuestros datos.

Lo declararemos en nuestro archivo test.js, y ahora es cuando comenzaremos a utilizar ese archivo, así que asegúrate de importarlo en home.html. El objeto únicamente necesita tener 3 arreglos vacíos inicialmente: uno de users, uno de tasks y uno de tags:

Entonces, comenzando con los usuarios, crearemos las funciones necesarias para ir llenando nuestra variable “data” (el objeto de información). Todas estas funciones las declararemos en nuestro archivo users_controllers.js. A continuación se explican las funciones:

• createUser(name, email, password): Ya puedes imaginar lo que hace esta función.

Recibe la información que tendrá el usuario, y creas un nuevo objeto de tipo User.

Haces push de dicho objeto en tu variable “data”, dentro de “users” (data.users.push(obj)), y listo. No necesita ningún valor de retorno.

IMPORTANTE: A partir de esta función ya tienes información de la cantidad de usuarios, por lo que ya deberías de actualizar tu función getNextUserID() para que, ahora, en vez de retornar “1”, te regrese el ID del consecutivo que corresponda.

• getUserById(id): Como parámetro, recibe un ID. Dado que los usuarios que creas ya tienen un ID correcto, debes de poder utilizarla con el método find buscando el ID. Si no encuentra el usuario, debe regresar “404 - User not found”. Si lo encuentra, su valor de retorno debe ser el usuario encontrado.

• searchUsers(attribute, value): Recibe un atributo (string) y un valor. Utilizando filter, debes de regresar un arreglo con los Users que cuenten con un atributo que incluya el valor recibido (puedes utilizar includes() para verificar que sí lo incluya).

NOTA: Si recibe un atributo que no existe, debe de lanzar una excepción indicándolo. Puedes

verificar si el atributo existe con User.hasOwnProperty(attribute). ATENTO, debes de
poder filtrar también por el atributo de “joined_at”, y utilizar una fecha. En tus pruebas será redundante, pues “joined_at” te regresará la misma fecha de hoy, siempre. Pero hacer esto te será útil para más adelante filtrar en Tasks.
• getAllUsers(): Únicamente te regresa tu arreglo de usuarios desde el objeto data.
• updateUser(id, obj_new_info): Recibe el id de un usuario a actualizar, y un objeto
con la información que se quiere actualizar. Si el usuario se actualiza correctamente,
debe de regresar “true”:
Considera que si no encuentra el usuario (se recibe un ID inexistente), debe arrojar
una excepción indicándolo. Si recibe un objeto con atributos que no existan, debe
ignorar estos atributos, pero SÍ ACTUALIZAR los que existan. Si no existe ningún
atributo y no actualizó nada de la información, debe lanzar una excepción indicándolo.
• deleteUser(id): Recibe un id de un usuario a eliminar de tu arreglo en data.users. Si
el usuario no existe, lanzar una excepción indicándolo.
IMPORTANTE: Dado que ahora ya tienes usuarios en tu objeto de información, modifica el
setter de email para que ahora sí valide que el email no se repita dentro de tu objeto de
información, para que no haya dos usuarios con el mismo correo. También modifica el setter
de “owner” en el modelo de Task para asegurarte de que reciba un valor de un usuario
existente.
Con esto, ya tienes todas las funciones necesarias para manipular la información de los
usuarios en tu objeto de información. DEBES DE PODER VALIDAR QUE FUNCIONAN
probándolas desde la consola. Ahora, haremos lo mismo para las Task y Tags.

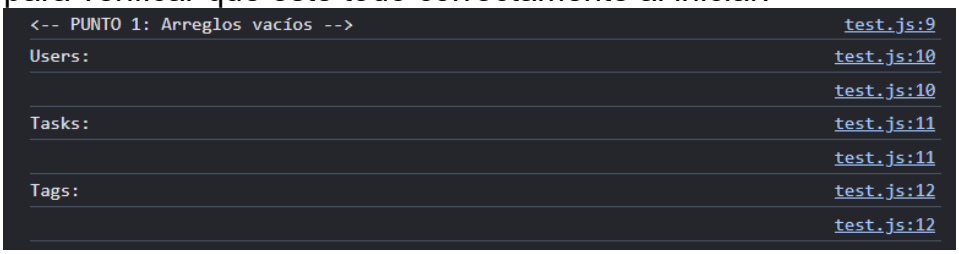
CRUD de Tareas y Etiquetas

Acabamos de hacer que, en nuestro objeto de información, puedas Crear, Leer, Actualizar y
Eliminar usuarios. A esto se le conoce como un CRUD (Create, Read, Update, Delete). Haremos
exactamente lo mismo para las Tareas y para las Etiquetas.
En este caso, las funciones de las Tareas serán en el archivo

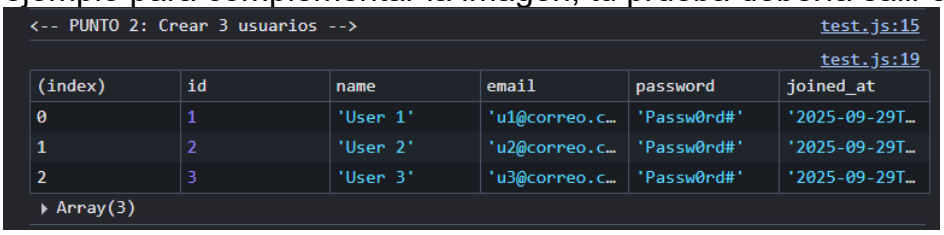
tasks_controller.js, mientras que
las de las etiquetas serán en el archivo tags_controller.js. No olvides importarlos en tu
home.html. Las funciones deben hacer exactamente lo mismo que las de users_controller.js,
pero ahora adaptadas a funcionar almacenando la información de tags y de tasks, con sus
respectivos atributos y las validaciones que apliquen.
Algunas cosas que debes de considerar:
• Al crear una Task, el arreglo de Tags que recibe (un arreglo de IDs: [1,5,2...]) debe de
tener únicamente Tags que existan, por lo que debes de modificar el setter para valide
no solo que se reciba un arreglo en el atributo “tags”, sino para que valide también
que las tags que está recibiendo existan.
• Para las Task, en la función “searchTasks”, debes de validar la fecha correctamente. Si
el atributo que estás recibiendo corresponde a algo relacionado con la fecha, debe de
funcionar bien.
• No se deben de poder eliminar etiquetas que estén asignadas a una tarea. Para esto,
en la función deleteTag, debes de hacer una validación de que la etiqueta a eliminar
no esté en ninguna tarea (será un for dentro de otro for). Si se da el caso, debes de
arrojar una excepción indicando que no se pueden eliminar etiquetas asignadas a
tareas, E INDICAR A QUÉ TAREA ESTÁN ASIGNADAS:
• Bajo esta misma lógica, no se debe poder eliminar un usuario si tiene tareas asignadas.
Valida y muestra excepciones de acuerdo a como aplique.
• ADICIONAL: En el controlador de tareas, debe de haber una función
findTasksByTag(array). Recibe un arreglo de IDs de etiquetas ([1,2,5,6...]), y debe de
regresarte las tareas que contengan esas etiquetas.
De nuevo, PRUEBA que todo funcione desde la consola. Debes poder llamar a todas tus
funciones de los controladores y verificar que tu objeto de información se actualiza
correctamente. Aunque para revisar, deberás de tener todos los ejemplos claros, como
veremos ahora.

Pruebas y Ejemplos

En test.js andaremos llamar nuestras diferentes funciones para validar que todo se ejecute según lo planeado.
Todas tus pruebas deben tener un console.log claro que las separe y las identifique correctamente. Si no se aprecia claramente el console.log y el console.table que corresponde, SE TE DESCONTARÁN LOS PUNTOS DE ESA PRUEBA. Debe haber un console.table mostrando la información necesaria para que la prueba se vea exitosa. Todas las pruebas que se relacionen con el CRUD deben de UTILIZAR LAS FUNCIONES DE LOS CONTROLADORES. No puedes manipular directamente tu objeto de información desde test.js. REVISARÉ EL ARCHIVO, y si encuentro que en alguna prueba manipulas directamente tu variable “data” (además de la declaración), SE RESTARÁN 30PTS a tu calificación de la práctica, sin preguntas. A continuación, te dejo un ejemplo de cómo deben de ser tus pruebas (DEBEN INCLUIR EL SEPARADOR Y ESTAR BIEN ESTRUCTURADAS):
Y las pruebas que deben de hacer son las siguientes:
1. PUNTO 1: Imprime tus arreglos del objeto de información vacíos. Esto será para verificar que esté todo correctamente al iniciar.

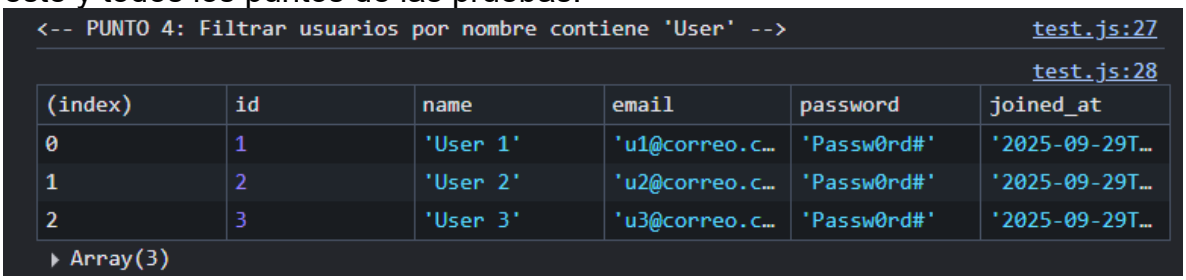


2. PUNTO 2: Crea 3 usuarios y muéstralos creados correctamente. Como ejemplo para complementar la imagen, tu prueba debería salir así:



3. PUNTO 3: Muestra la información del usuario con el ID = 2.

4. PUNTO 4: Muestra el resultado de filtrar usuarios por nombre. No lo olvides: debes incluir el console.table que aplique en el contexto correspondiente, para este y todos los puntos de las pruebas.

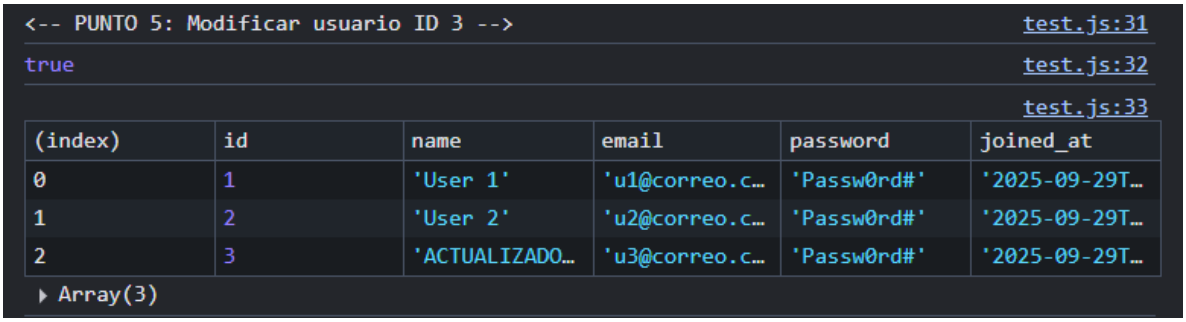


```
<-- PUNTO 4: Filtrar usuarios por nombre contiene 'User' -->   test.js:27
                                                              test.js:28
```

(index)	id	name	email	password	joined_at
0	1	'User 1'	'u1@correo.c…	'Passw0rd#'	'2025-09-29T…
1	2	'User 2'	'u2@correo.c…	'Passw0rd#'	'2025-09-29T…
2	3	'User 3'	'u3@correo.c…	'Passw0rd#'	'2025-09-29T…

```
▸ Array(3)
```

5. PUNTO 5: Modifica el usuario con el ID 3 para que ahora se llame “ACTUALIZADORX”. No olvides mostrar el console.table con tu información actualizada.

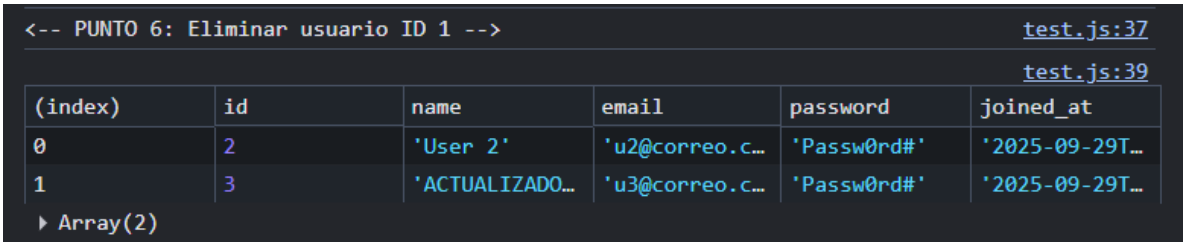


```
<-- PUNTO 5: Modificar usuario ID 3 -->   test.js:31
true                                      test.js:32
                                          test.js:33
```

(index)	id	name	email	password	joined_at
0	1	'User 1'	'u1@correo.c…	'Passw0rd#'	'2025-09-29T…
1	2	'User 2'	'u2@correo.c…	'Passw0rd#'	'2025-09-29T…
2	3	'ACTUALIZADO…	'u3@correo.c…	'Passw0rd#'	'2025-09-29T…

```
▸ Array(3)
```

6. PUNTO 6: Elimina el usuario con el ID 1.

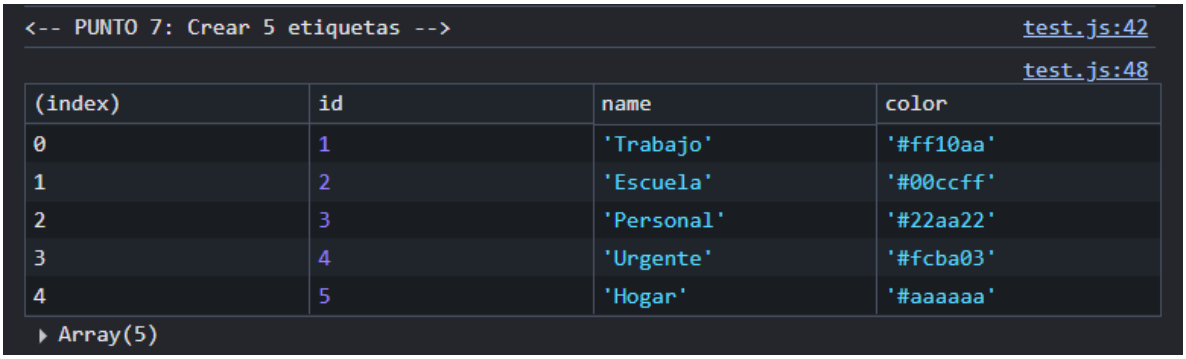


```
<-- PUNTO 6: Eliminar usuario ID 1 -->   test.js:37
                                         test.js:39
```

(index)	id	name	email	password	joined_at
0	2	'User 2'	'u2@correo.c…	'Passw0rd#'	'2025-09-29T…
1	3	'ACTUALIZADO…	'u3@correo.c…	'Passw0rd#'	'2025-09-29T…

```
▸ Array(2)
```

7. PUNTO 7: Crea 5 etiquetas diferentes. Recuerda mostrarlas con el console.table.



```
<-- PUNTO 7: Crear 5 etiquetas -->   test.js:42
                                     test.js:48
```

(index)	id	name	color
0	1	'Trabajo'	'#ff10aa'
1	2	'Escuela'	'#00ccff'
2	3	'Personal'	'#22aa22'
3	4	'Urgente'	'#fcba03'
4	5	'Hogar'	'#aaaaaa'

```
▸ Array(5)
```

8. PUNTO 8: Modifica la etiqueta con ID 4 para que ahora se llame “ETIQUETADORX” y que tenga el color “#fcba03”.

```
<-- PUNTO 8: Modificar etiqueta ID 4 -->                    test.js:51
true                                                        test.js:52
                                                            test.js:53
```

(index)	id	name	color
0	1	'Trabajo'	'#ff10aa'
1	2	'Escuela'	'#00ccff'
2	3	'Personal'	'#22aa22'
3	4	'ETIQUETADORX'	'#fcba03'
4	5	'Hogar'	'#aaaaaa'

▸ Array(5)

9. PUNTO 9: Elimina la etiqueta con ID 2.

```
<-- PUNTO 9: Eliminar etiqueta ID 2 -->                     test.js:56
                                                            test.js:58
```

(index)	id	name	color
0	1	'Trabajo'	'#ff10aa'
1	3	'Personal'	'#22aa22'
2	4	'ETIQUETADORX'	'#fcba03'
3	5	'Hogar'	'#aaaaaa'

▸ Array(4)

10. PUNTO 10: Crea 7 tareas diferentes. TODAS deben de tener al menos 2 etiquetas existentes asignadas.

```
<-- PUNTO 10: Crear 7 tareas (>=2 tags) -->                 test.js:61
                                                            test.js:70
```

(index)	id	title	descripti…	due_date	owner	status	tags
0	1	'Tarea 1'	'Desc 1'	'2025-10-…	2	'A'	Array(2)
1	2	'Tarea 2'	'Desc 2'	'2025-10-…	2	'A'	Array(2)
2	3	'Tarea 3'	'Desc 3'	'2025-10-…	3	'C'	Array(2)
3	4	'Tarea 4'	'Desc 4'	'2025-10-…	3	'F'	Array(2)
4	5	'Tarea 5'	'Desc 5'	'2025-10-…	2	'A'	Array(2)
5	6	'Tarea 6'	'Desc 6'	'2025-10-…	3	'A'	Array(2)
6	7	'Tarea 7'	'Desc 7'	'2025-10-…	2	'A'	Array(3)

▸ Array(7)

11. PUNTO 11: Modifica la tarea con ID 5 para que no tenga etiquetas. No olvides mostrarlo con el console.table (debes hacer esto con todos los puntos, pero te lo recuerdo por si acaso).

```
<-- PUNTO 11: Tarea ID 5 sin etiquetas -->                  test.js:73
                                                            test.js:75
```

(index)	id	title	descripti…	due_date	owner	status	tags
0	1	'Tarea 1'	'Desc 1'	'2025-10-…	2	'A'	Array(2)
1	2	'Tarea 2'	'Desc 2'	'2025-10-…	2	'A'	Array(2)
2	3	'Tarea 3'	'Desc 3'	'2025-10-…	3	'C'	Array(2)
3	4	'Tarea 4'	'Desc 4'	'2025-10-…	3	'F'	Array(2)
4	5	'Tarea 5'	'Desc 5'	'2025-10-…	2	'A'	Array(0)
5	6	'Tarea 6'	'Desc 6'	'2025-10-…	3	'A'	Array(2)
6	7	'Tarea 7'	'Desc 7'	'2025-10-…	2	'A'	Array(3)

▸ Array(7)

12. PUNTO 12: Modifica la tarea con ID 1 y con ID 4 para que su descripción diga “Dorx Task”.

```
<-- PUNTO 12: Tareas ID 1 y 4 con descripción 'Dorx Task' -->   test.js:78
                                                                test.js:81
```

(index)	id	title	descripti…	due_date	owner	status	tags
0	1	'Tarea 1'	'Dorx Tas…	'2025-10-…	2	'A'	Array(2)
1	2	'Tarea 2'	'Desc 2'	'2025-10-…	2	'A'	Array(2)
2	3	'Tarea 3'	'Desc 3'	'2025-10-…	3	'C'	Array(2)
3	4	'Tarea 4'	'Dorx Tas…	'2025-10-…	3	'F'	Array(2)
4	5	'Tarea 5'	'Desc 5'	'2025-10-…	2	'A'	Array(0)
5	6	'Tarea 6'	'Desc 6'	'2025-10-…	3	'A'	Array(2)
6	7	'Tarea 7'	'Desc 7'	'2025-10-…	2	'A'	Array(3)

```
▸ Array(7)
```

13. PUNTO 13: Filtra para mostrar todas las tareas que incluyan “Dorx” en su descripción.

```
<-- PUNTO 13: Filtrar tareas con 'Dorx' en descripción -->   test.js:84
                                                             test.js:85
```

(index)	id	title	descripti…	due_date	owner	status	tags
0	1	'Tarea 1'	'Dorx Tas…	'2025-10-…	2	'A'	Array(2)
1	4	'Tarea 4'	'Dorx Tas…	'2025-10-…	3	'F'	Array(2)

```
▸ Array(2)
```

14. PUNTO 14: Filtra para mostrar tareas a partir de alguna etiqueta (o varias etiquetas).

```
<-- PUNTO 14: Filtrar tareas por etiqueta(s) -->   test.js:88
Por etiqueta 1:                                    test.js:89
                                                   test.js:90
```

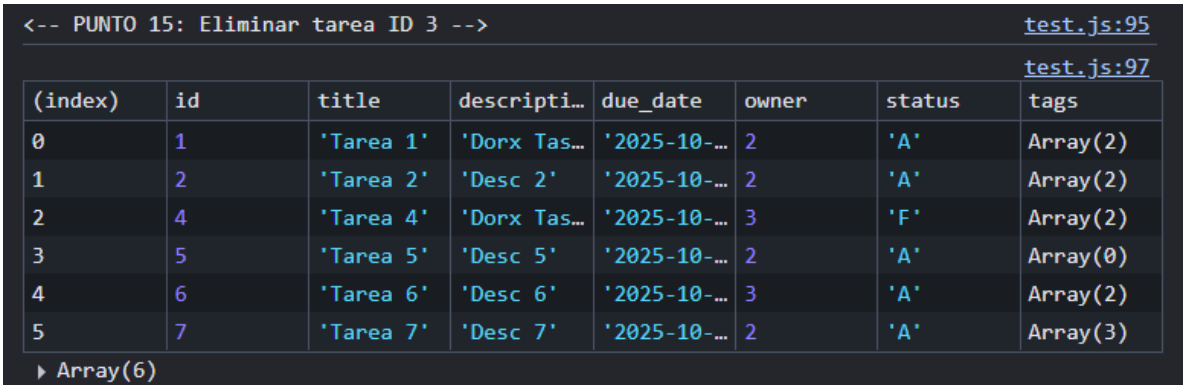
(index)	id	title	descripti…	due_date	owner	status	tags
0	1	'Tarea 1'	'Dorx Tas…	'2025-10-…	2	'A'	Array(2)
1	3	'Tarea 3'	'Desc 3'	'2025-10-…	3	'C'	Array(2)
2	4	'Tarea 4'	'Dorx Tas…	'2025-10-…	3	'F'	Array(2)
3	7	'Tarea 7'	'Desc 7'	'2025-10-…	2	'A'	Array(3)

```
▸ Array(4)
findTasksByTag([1,5]) (ANY):   test.js:91
                               test.js:92
```

(index)	id	title	descripti…	due_date	owner	status	tags
0	1	'Tarea 1'	'Dorx Tas…	'2025-10-…	2	'A'	Array(2)
1	3	'Tarea 3'	'Desc 3'	'2025-10-…	3	'C'	Array(2)
2	4	'Tarea 4'	'Dorx Tas…	'2025-10-…	3	'F'	Array(2)
3	6	'Tarea 6'	'Desc 6'	'2025-10-…	3	'A'	Array(2)
4	7	'Tarea 7'	'Desc 7'	'2025-10-…	2	'A'	Array(3)

```
▸ Array(5)
```

15. PUNTO 15: Elimina la tarea con ID 3.



16. PUNTO 16: Adicionalmente, correré pruebas en la consola para validar el uso de excepciones. Estas las haré yo, ya que una excepción detiene el código que se tenga después. Probaré varias cosas al azar, como crear un usuario con contraseña pequeña, eliminar una etiqueta asignada a una tarea, duplicar correos, etc.
17. PUNTO 16: Reporte.

Evaluación

Sección Puntuación
PUNTO 1 0 puntos *
PUNTO 2 9 puntos *
PUNTO 3 3 puntos *
PUNTO 4 4 puntos *
PUNTO 5 5 puntos *
PUNTO 6 4 puntos *
PUNTO 7 9 puntos *
PUNTO 8 5 puntos *
PUNTO 9 4 puntos *
PUNTO 10 9 puntos *
PUNTO 11 5 puntos *
PUNTO 12 5 puntos *
PUNTO 13 5 puntos *
PUNTO 14 5 puntos *
PUNTO 15 4 puntos *
PUNTO 16 20 puntos *
PUNTO 17 4 puntos *
Entregables:
• Recuerda entregar un zip de tu carpeta P02, para que yo pueda simplemente descomprimirlo y correr con live server tu home.html con los archivos js importados.
VALIDA QUE TU ENTREGABLE FUNCIONE. Recuerda, lo correré con Live Server y no entraré a revisar código más allá del test.js.
• Debes además entregar un reporte (PDF) con las evidencias y la rúbrica contestada de acuerdo a lo que se alcanzó a entregar.

• En el reporte añadir conclusiones, cosas por mejorar, temas que costaron trabajo, etc.

Conclusión:

En cosas por mejorar, tal vez hubiera sido bueno analizar otros diseños de algunas partes de código para ver su eficiencia y tal vez sin tantos errores.

Fue una actividad llena de imprevistos, ya al final no me querían correr los modelos y los controladores con “live server”, aparecía un tipo de error que decía algo parecido a que los archivos de los “.js” no se habían encontrado, pero al parecer si se podían ver en lo local aunque a veces también fallaba, también para algunos pasos por ejemplo, de las etiquetas y otros atributos no querían leerse o editarse, en la parte final de los puntos de las etiquetas use inteligencias artificiales para ver si algo estaba ocasionando un error o un conflicto, pero querían que editara gran parte de todo los códigos y eso pudo haber afectado a más cosas, así que mejor no utilice esas recomendaciones.

Al final tuve que analizar línea por línea y también hacer validaciones y pruebas para que si se estuvieran registrando los datos correctamente.

En Canvas debe de haber dos archivos: tu reporte en PDF y un zip con tu práctica.