Persistencia

La persistencia es la capacidad de un sistema para conservar datos o estados más allá de la ejecución del programa o el apagado del dispositivo. En el desarrollo de aplicaciones web, los navegadores modernos ofrecen mecanismos de almacenamiento local que permiten gestionar tanto pequeñas configuraciones como grandes volúmenes de datos estructurados. Estos mecanismos también facilitan la implementación de cachés para reducir la latencia y optimizar el uso del ancho de banda, factores clave para la escalabilidad y la experiencia del usuario.

Los objetos *localStorage*, *sessionStorage* e *IndexedDB* permiten almacenar datos de forma persistente en el navegador, desde configuraciones básicas hasta conjuntos de datos complejos. Además, *Cache Storage*, en combinación con *Service Workers*, optimiza la carga de recursos y mejora el funcionamiento offline.

Cada uno de estos mecanismos ofrece ventajas específicas según el contexto de uso. A continuación, exploraremos algunos de ellos:

SessionStorage

El objeto window.sessionStorage[1] es creado por el navegador de forma independiente para cada pestaña. Los datos persisten mientras la pestaña esté abierta y se eliminan al cerrarla. Los datos almacenados persisten tras recargar o restaurar la pestaña. Si una pestaña se duplica, recibe una copia independiente del sessionStorage de la original, sin sincronización entre ambas. El almacenamiento es exclusivo de cada pestaña, incluso si se accede a la misma URL en otra ventana o pestaña. sessionStorage tiene una capacidad de ~5MB y almacena pares clave-valor como strings.

Algunas aplicaciones comunes incluyen:

- **Persistencia de filtros y opciones de visualización:** Permite recordar filtros y preferencias de visualización durante la sesión, evitando que el usuario deba configurarlos nuevamente tras recargar o navegar entre resultados.
- **Control de la navegación en SPA:** En aplicaciones de una sola página (SPA), sessionStorage puede almacenar el estado de la interfaz, como la página actual o el contenido de un carrito de compras.
- **Autenticación temporal:** Permite almacenar un token de autenticación tras el inicio de sesión, asegurando que la sesión se mantenga mientras la pestaña esté abierta.
- **Registro de interacciones en aplicaciones interactivas:** En juegos, encuestas u otras aplicaciones dinámicas, sessionStorage permite registrar interacciones en tiempo real sin depender de un backend para cada acción.

LocalStorage

El objeto window.localStorage[2] es creado por el navegador y es accesible desde todas las pestañas y ventanas que compartan el mismo origen. Las modificaciones en el almacenamiento son visibles en otras pestañas tras una recarga o mediante eventos de almacenamiento. Los datos almacenados persisten a través de recargas, restauraciones de pestañas y reinicios del navegador o del dispositivo, salvo que sean eliminados explícitamente. A diferencia de sessionStorage, donde los datos se limitan a la pestaña actual, localStorage es accesible desde cualquier pestaña o ventana del mismo origen y dispone de un espacio mayor ~50MB de almacenamiento. Almacena también pares clave-valor como strings.

Algunas aplicaciones comunes incluyen:

- **Almacenamiento de preferencias de usuario:** Permite guardar configuraciones como el tema visual o el idioma, asegurando su conservación tras cerrar el navegador.
- **Estados parciales de formularios:** Facilita el llenado de formularios recurrentes al evitar la pérdida de datos por cierres inesperados o recargas accidentales.

[1] https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage
[2] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

- **Trabajo offline:** Permite almacenar cambios temporalmente en aplicaciones que funcionan sin conexión y sincronizarlos con el backend más tarde.
- **Caché ligera para mejorar rendimiento:** Puede usarse para almacenar respuestas pequeñas del backend y reducir el tráfico con el servidor, considerando la limitación de 5MB.
- **Búfer de telemetría:** Permite registrar interacciones de usuario y eventos en localStorage y enviarlos al servidor en lotes para optimizar el análisis de datos.

Tanto sessionStorage como localStorage tienen una Web Storage API de acceso común que resulta similar a la interfaz de uso de los mapas clave-valor.

Método	Descripción
setItem(key, value)	Almacena un par clave-valor. El valor se guarda como una cadena (string)
getItem(key)	Recupera el valor asociado a una clave. Retorna null si la clave no existe.
removeItem(key)	Elimina un elemento de almacenamiento por su clave.
clear()	Elimina todos los elementos almacenados en localStorage o sessionStorage.
key(index)	Recupera el nombre de la clave en el índice especificado. Es útil cuando deseas iterar sobre todas las claves almacenadas.
length	Propiedad que devuelve el número de elementos almacenados en localStorage o sessionStorage.

SessionStorage como LocalStorage son síncronos, lo que significa que bloquean la ejecución de otro código JavaScript hasta que la operación se complete. Este comportamiento puede afectar el rendimiento, especialmente cuando se manejan grandes volúmenes de datos.

Por esta razón, es importante reducir las operaciones intensivas en sessionStorage o localStorage que impliquen grandes volúmenes de datos. Como regla general, se recomienda minimizar las operaciones síncronas para evitar bloqueos en la interfaz de usuario y mejorar la capacidad de respuesta de la aplicación.

Cookies

Una cookie[3] es una cadena de texto con un par clave-valor acompañada de atributos que determinan su comportamiento. Puede ser enviada desde el servidor mediante la cabecera HTTP Set-Cookie en las respuestas al cliente o creada en el cliente con document.cookie. Es posible definir múltiples cookies, cada una con sus propios atributos.

Formato de cookies (Cabecera HTTP/S)
`Set-Cookie: key=value; Attribute1; Attribute2; … AttributeN=value`

El atributo Expires o Max-Age determina si una cookie es de sesión, eliminándose al cerrar el navegador, o persistente, almacenándose en el dispositivo hasta su fecha de expiración. Los atributos Domain y Path controlan el ámbito de la cookie: el primero define el dominio válido, permitiendo subdominios, y el segundo restringe la cookie a una ruta específica, limitando su envío a solicitudes que coincidan con esa ruta. Atributos como HttpOnly impiden el acceso desde código JavaScript del cliente, Secure limita su transmisión a conexiones seguras, y SameSite regula el envío a terceros.

En todas las peticiones enviadas al servidor dentro del dominio y ruta especificados, las cookies almacenadas se incluyen automáticamente en la cabecera HTTP Cookie, sin intervención del usuario o scripts en el cliente, aunque para el caso de cookies creadas por el cliente, se deben cumplir restricciones adicionales: debe pertenecer al mismo dominio y su Path debe coincidir con la ruta de la solicitud. Además, si tiene SameSite=Strict, solo se enviará en navegaciones dentro del mismo sitio, mientras que con SameSite=Lax podrá incluirse en algunas solicitudes de terceros, como

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

enlaces. Si se establece Secure, solo será enviada en conexiones HTTPS. No puede configurarse HttpOnly, ya que este atributo solo puede ser definido por el servidor.

Para todos los casos, es importante destacar que las cookies tienen una limitación de 4KB de almacenamiento y se restringen a una cantidad máxima de 50 por dominio.

Algunas de las aplicaciones más comunes de las cookies son:

- **Gestión de sesiones**: Permiten mantener la autenticación de los usuarios en sitios web sin necesidad de que inicien sesión en cada página visitada. Se usan para almacenar identificadores de sesión que el servidor reconoce.
- **Personalización de la experiencia del usuario**: Se pueden utilizar para recordar preferencias del usuario, como el idioma seleccionado, el tema de la interfaz o configuraciones específicas de una aplicación web.
- **Seguimiento y análisis**: Muchas plataformas de análisis web utilizan cookies para rastrear la actividad del usuario en un sitio y recopilar datos sobre su comportamiento, lo que permite mejorar la navegación y optimizar el contenido.
- **Publicidad dirigida**: Los sistemas de publicidad emplean cookies para registrar las interacciones de los usuarios con anuncios y personalizar la publicidad mostrada en función de sus intereses.
- **Almacenamiento temporal de datos**: En sitios de comercio electrónico, las cookies permiten mantener productos en el carrito de compras incluso si el usuario abandona la página y regresa más tarde.
- **Seguridad y control de accesos**: Algunas cookies se usan para detectar actividades sospechosas, prevenir ataques de sesión y reforzar medidas de seguridad en autenticaciones en dos pasos o restricciones de acceso.

A continuación se presenta un ejemplo simplificado que refleja el tráfico de cookies entre un cliente y un servidor:

Ejemplo de primera petición realizada desde el cliente (Cabecera HTTP/S)

```
GET / HTTP/1.1
Host: example.com
```

Ejemplo de respuesta desde el servidor (Cabecera HTTP/S)

```
HTTP/1.1 200 OK
Set-Cookie: session_id=abc123; Path=/; HttpOnly; Max-Age=3600; Secure; SameSite=Strict
Set-Cookie: user_pref=dark_mode; Path=/; Domain=example.com; Max-Age=31536000; Secure; SameSite=Lax
Set-Cookie: shopping_cart=items_12345; Path=/cart; Max-Age=86400; HttpOnly
Content-Type: text/html; charset=UTF-8
```

Ejemplo de la siguiente petición realizada por el cliente (Cabecera HTTP/S)

```
GET /cart HTTP/1.1
Host: example.com
Cookie: session_id=abc123; user_pref=dark_mode; shopping_cart=items_12345
```

Como se puede observar, al recibir la primera respuesta del servidor, el cliente procesa la cabecera y el navegador construye las cookies en base a los atributos proporcionados. Sin embargo, en la siguiente solicitud realizada por el cliente (a la ruta /cart), sólo se envían los pares clave-valor de las cookies válidas para esa ruta, y no se reenvían los atributos recibidos, ya que es el servidor quien las define.

A continuación se presenta una tabla que amplía el significado de los atributos posibles para la definición de cookies:

Función	Atributo	Descripción
Envío de datos	clave=valor	Es el par clave y valor de la cookie. El nombre debe ser único dentro del mismo dominio.
Control del tiempo de validez	Expires	Define la fecha y hora exacta en que la cookie debe expirar. Si no se especifica, la cookie será de sesión, es decir, se eliminará cuando se cierre el navegador. Debe ser una fecha en formato UTC.
	Max-Age	Define la duración en segundos desde el momento de su creación hasta su expiración. Es una alternativa a

		Expires. Si se especifica, Expires es ignorado.
Control de dominio y rutas	Path	Define el alcance dentro de la aplicación web en la que la cookie es válida. Es decir, especifica qué rutas de la URL pueden acceder a la cookie cuando se realizan solicitudes al servidor o cuando se accede a la cookie desde JavaScript en el navegador (si la cookie no es HttpOnly). Si Path se establece en /, la cookie será accesible en todas las rutas del dominio. Por ejemplo, si se establece en /home, será sólo válida para el *dominio.com/home* como también todas las sub-rutas interiores: *dominio.com/home/sub1*.
	Domain	Define el dominio para el cual la cookie es válida. Si no se especifica, la cookie será válida sólo para el dominio que la creó (y no para subdominios). Si se especifica, permite que subdominios del dominio definido también tengan acceso a la cookie.
Seguridad	Secure	Indica que la cookie solo debe enviarse a través de una conexión segura HTTPS. No se enviará en solicitudes HTTP no seguras.
	HttpOnly	Evita que la cookie sea accesible desde JavaScript. Esto ayuda a protegerla contra ataques de tipo XSS (Cross-Site Scripting), ya que JavaScript no podrá leer la cookie.
	SameSite	Controla si la cookie debe ser enviada en solicitudes de sitios de terceros. Es útil para prevenir ataques CSRF (Cross-Site Request Forgery) Strict: La cookie solo se enviará en solicitudes del mismo origen (el mismo dominio). Lax: La cookie se enviará en algunas solicitudes de terceros (por ejemplo, en un enlace de navegación entre sitios). None: La cookie se enviará en todas las solicitudes, incluyendo solicitudes de terceros. Debe estar acompañada de Secure para ser válida en HTTPS.
Prioridad (Opcional)	Priority	Especifica la prioridad de la cookie. Algunos navegadores permiten definir si una cookie debe ser tratada como de alta, baja o normal prioridad en el almacenamiento y envío. Este atributo no es ampliamente utilizado. Valores posibles: High, Low, Medium.

IndexedDB

El objeto de almacenamiento *window.IndexedDB*[4] permite el almacenamiento estructurado de datos en el cliente. Se trata de una instancia de conexión a una base de datos NoSQL (clave-valor) indexada, transaccional y asincrónica. A diferencia de *localStorage* y *sessionStorage*, está diseñado para manejar grandes volúmenes de información de manera eficiente. La información guardada en *IndexedDB* persiste incluso después de cerrar el navegador y es accesible desde todas las pestañas y ventanas del mismo origen. Además, admite almacenamiento de objetos complejos, índices y consultas avanzadas, lo que lo hace adecuado para aplicaciones web con necesidades de almacenamiento más sofisticadas. En IndexedDB, la capacidad de almacenamiento ronda el ~60% del espacio libre del dispositivo. Algunas de sus aplicaciones más comunes son:

- **Diseño de caché de gran capacidad:** IndexedDB no impone restricciones de tamaño, lo que lo convierte en la mejor opción para caché de alto volumen, reduciendo significativamente el tráfico de red. Su uso es común en videojuegos, donde almacena grandes volúmenes de datos estáticos como texturas, modelos y sonidos, así como en el almacenamiento de respuestas invariables de API, como información geográfica y mapas.

- **Aplicaciones offline y autónomas:** Permite que una aplicación funcione sin conexión al persistir localmente los datos necesarios. Además de los videojuegos, facilita el desarrollo de herramientas de edición multimedia, ofimática y otras aplicaciones que tradicionalmente requieren software de escritorio.

- **Sincronización en segundo plano:** Permite trabajar con una réplica local de datos del servidor y sincronizarla mediante WebWorkers periódicamente o tras recuperar la conexión a internet.

- **Telemetría y gestión de sesiones:** IndexedDB posibilita el almacenamiento y análisis de grandes volúmenes de datos sobre la interacción del usuario, facilitando el monitoreo, la recolección de diagnósticos y el envío de información a un servidor para análisis posterior.

- **Aplicaciones avanzadas:** IndexedDB abarca todas las capacidades de sessionStorage y localStorage, además de permitir aplicaciones más complejas como análisis de datos en cliente, machine learning, almacenamiento de claves criptográficas, blockchain, simulación de bases de datos y ejecución de máquinas virtuales.

[4] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

Para trabajar con IndexedDB, generalmente se comienza con la invocación de `indexedDB.open("SomeDatabaseName", 1)`, que es el método utilizado para abrir o crear la base de datos. El primer argumento corresponde al nombre de la base de datos, ya sea existente o nueva. Dado que la operación no es inmediata debido al comportamiento asincrónico, `open()` devuelve un objeto `IDBOpenDBRequest`, que se encarga de disparar eventos de éxito (`success`) y error (`error`), y de ejecutar los manejadores que tengan asignados. El segundo argumento es un número entero de 64 bits (`int64`) que indica la versión del esquema subyacente (la estructura de los objetos almacenados). Si la base de datos no existe bajo el nombre proporcionado, o si se proporciona un número de versión superior, el objeto `IDBOpenDBRequest` dispara un evento `onupgradeneeded` y ejecuta su manejador asignado, con el propósito de asignar o actualizar el esquema de datos. El verdadero acceso a la base de datos se realiza a través del uso de la propiedad `result` proporcionada por el objeto de evento recibido por el manejador correspondiente.

En caso de definir todos los manejadores para las diversas situaciones resultaría algo como a continuación:

```
let dbInstance = null;

function onDatabaseConnectionError(event){
   // Do something..
};

function onDatabaseConnectionSuccess(event){
  dbInstance = event.target.result;
  // Do something..
};

function onDatabaseUpgrade(event){
 // Do something..
}

const request = indexedDB.open("DatabaseExample", 1);

request.onerror = onDatabaseConnectionError;
request.onsuccess = onDatabaseConnectionSuccess;
request.onupgradeneeded = onDatabaseUpgrade;
```

El evento `onsuccess` está directamente relacionado con la "inicialización" de la base de datos, pero específicamente se refiere a la finalización exitosa de la operación de apertura. En este evento, la base de datos ya ha sido abierta correctamente (o creada, si no existía) y está lista para ser utilizada. Nótese que el manejador asignado en este caso asocia la referencia dbInstance a la instancia real de la base de datos para posibilitar futuras operaciones.

Dado que IndexedDB es una base de datos transaccional, todas las operaciones de manipulación de datos (lecturas y escrituras) se agrupa en una única unidad de trabajo (transacción), asegurando que todas las modificaciones se realicen de manera atómica (todas o ninguna). Las transacciones en IndexedDB tiene únicamente tres estados posibles complete (satisfactoria), error (falla) y abort (abortada), también son asincrónicas y su gestión descansa sobre el manejo de eventos.

El objeto `window.indexedDB` hereda de `EventTarget`[5], lo que permite gestionar eventos mediante su interfaz base y asignar múltiples manejadores cuando sea necesario.

Ejemplo de asignación de manejadores de eventos a través del prototipo EventTarget

```
const request = indexedDB.open("DatabaseExample", 1);

function onSuccessEventHandlerA(event){
console.log("Executing EventHandler A");
};

function onSuccessEventHandlerB(event){
console.log("Executing EventHandler B");
};

request.addEventListener("success", onSuccessEventHandlerA );
request.addEventListener("success", onSuccessEventHandlerB );
```

[5] https://developer.mozilla.org/en-US/docs/Web/API/EventTarget

Los eventos tienen exactamente los mismos nombres que las propiedades manejadoras de eventos sin el prefijo "on". Ej: `onupgradeneeded` sería `upgradeneeded` como argumento del método **`addEventListener`**.

A continuación se presenta un ejemplo extendido que integra lo explicado anteriormente involucrando el uso de transacciones:

Ejemplo de uso de IndexedDB

```
let dbInstance = null;

function onDatabaseUpgrade(event) {
    const db = event.target.result;
    if (!db.objectStoreNames.contains("user")) {
        db.createObjectStore("user", { keyPath: "id" });
    }
}

function onDatabaseConnectionSuccess(event) {
    dbInstance = event.target.result;
    console.log("Database is ready");
}

function onDatabaseConnectionError() {
    console.error("Database Connection Error");
}

function onUserAdded() {
    console.log("User added successfully");
}

function onUserAddError() {
    console.log("Cannot add user");
}

function onUserAddAbort() {
    console.log("User add operation aborted/cancelled");
}

function addUser(usuario) {
    if (!dbInstance) {
        console.error("Database not initialized yet");
        return;
    }
    const transaction = dbInstance.transaction("user", "readwrite");
    transaction.oncomplete = onUserAdded;
    transaction.onerror = onUserAddError;
    transaction.onabort = onUserAddAbort;
    const store = transaction.objectStore("user");
    store.add(usuario);

}

function main() {
    const request = indexedDB.open("DatabaseExample", 1);
    request.onupgradeneeded = onDatabaseUpgrade;
    request.onsuccess = onDatabaseConnectionSuccess;
    request.onerror = onDatabaseConnectionError;
}

main();
```

El ejemplo presentado es muy acotado en funcionalidad, pero ilustra cómo crear una base de datos y efectuar una operación de escritura mediante el uso de transacciones. Por último, se presenta un resumen a modo de hoja de datos que explica los eventos y métodos más utilizados de IndexedDB en el contexto de aplicación sobre requerimientos del tipo CRUD (create,read,update,delete):

Interacción con la base de datos

indexedDB.open(name, version)	Este método se utiliza para abrir (o crear) una base de datos. Devuelve un objeto IDBOpenDBRequest que permite gestionar el proceso de apertura y manejar los eventos asociados (como onsuccess y onupgradeneeded).
Manejadores de eventos en la apertura/creación de base de datos	
onupgradeneeded	Se dispara cuando la base de datos necesita ser creada o actualizada, es decir, cuando no existe o cuando se especifica una versión superior a la actual. Aquí se definen o actualizan los esquemas de la base de datos (por ejemplo, creando object stores o índices).
onsuccess	Se dispara cuando la base de datos se abre correctamente. El objeto event.target.result contiene la instancia de IDBDatabase que es la que permite la interacción con la base de datos. Este evento marca la finalización de la etapa de "inicialización" de la base de datos y se usa para continuar con operaciones adicionales.
onerror	Se dispara cuando ocurre un error al intentar abrir la base de datos (por ejemplo, si no se otorgan los permisos para acceder a IndexedDB o si hay un problema técnico). Se usa para manejar el fallo y notificar al usuario eventualmente.
Transacciones en la base de datos	
IDBDatabase.createObjectStore(name, options)	Se usa dentro del evento onupgradeneeded para crear un nuevo object store (almacén de objetos) en la base de datos. Se especifica un nombre y, opcionalmente, un objeto de configuración (como keyPath para definir la clave primaria).
IDBObjectStore.add(value)	Permite agregar un valor a un object store. Si el valor ya existe, la operación fallará (si no se está usando un keyPath único o un índice). Se usa dentro de una transacción con el modo "readwrite".
IDBDatabase.transaction(storeNames, mode)	Crea una transacción en la base de datos. El primer parámetro es una lista de object stores involucrados en la transacción, y el segundo parámetro indica el modo de la transacción ("readonly" o "readwrite"). Devuelve un objeto IDBTransaction.
IDBTransaction.objectStore(name)	Este método se usa para obtener un object store dentro de una transacción. Se puede usar para realizar operaciones de lectura y escritura sobre el object store especificado.
IDBObjectStore.get(key)	Recupera un valor desde un object store utilizando una clave. Se usa dentro de una transacción de solo lectura (readonly) y devuelve el valor asociado con la clave proporcionada.
IDBObjectStore.getAll()	Recupera todos los objetos almacenados en un object store. Es útil cuando quieres obtener todos los elementos de un store sin filtrarlos por clave. Se usa dentro de una transacción de solo lectura.
IDBObjectStore.put(value)	Similar a add, pero permite actualizar un objeto existente si ya tiene la misma clave primaria (keyPath). Si el objeto no existe, se agrega como nuevo. Se usa dentro de una transacción con el modo "readwrite".
IDBObjectStore.delete(key)	Elimina un objeto del object store utilizando la clave primaria del objeto. Es una operación destructiva y se utiliza dentro de una transacción de escritura.
IDBObjectStore.delete(key)	Elimina un objeto del object store utilizando su clave primaria. Se usa dentro de una transacción de tipo "readwrite" y elimina el objeto correspondiente.
IDBObjectStore.clear()	Elimina todos los objetos de un object store, dejando el store vacío. Esta operación es destructiva y se usa dentro de una transacción de escritura.
IDBTransaction.abort()	Aborta una transacción en curso. Si se llama, todas las operaciones realizadas en la transacción se deshacen. Es útil en caso de error o necesidad de cancelación.
IDBObjectStore.count()	Cuenta el número de objetos en un object store sin obtener los datos. Se usa para obtener el número total de elementos en un store y se realiza dentro de una transacción de solo lectura.
IDBObjectStore.openCursor()	Abre un cursor en un object store para recorrer sus elementos de forma secuencial. Puedes usarlo para realizar iteraciones personalizadas o filtradas sobre los datos de un store.
Manejadores de evento de las transacciones	
oncomplete	Se dispara cuando una transacción se completa con éxito. Indica que todas las operaciones dentro de la transacción se han ejecutado correctamente. Se usa comúnmente para realizar acciones adicionales, como mostrar un mensaje de éxito.

onabort	Se dispara cuando una transacción es abortada por ejecución directa de transaction.abort() o de forma espontánea. Se usa para manejar situaciones donde se deshacen las operaciones realizadas en la transacción (rollback).
onerror	Se dispara cuando ocurre un error durante la transacción, como un problema al intentar leer o escribir en el object store. Permite gestionar errores específicos de la transacción.

Ejercitación

Hasta este punto nuestro programa es un prototipo básico o borrador. Todas las estructuras de datos funcionan en la memoria del programa. Por lo tanto, los datos existen únicamente cuando el programa se inicia y se destruyen cuando el programa termina su ejecución.

1. Desarrolle una solución en donde la persistencia esté resuelta empleando la API de WebStorage. Utilice SessionStorage y LocalStorage. Ejecute ambas opciones y determine ¿Cuál/es son las diferencias que percibe?

2. Realice el equivalente de la resolución del punto 1, mediante el uso de Cookies.

3. Desarrolle una solución alternativa en donde resuelva la persistencia a través de IndexedDB. ¿Qué diferencias nota respecto al punto 1 y 2?

4. Elabore una tabla comparativa sobre los 4 mecanismos de persistencia del lado cliente y establezca las ventajas/desventajas y casos de uso que son apropiados (definir con claridad en qué contexto uno es más apropiado que otro).

5. Extienda el programa de modo tal que cualquier acción que produzca un impacto en el stock de los artículos, quede registrada en una estructura de datos persistente adecuada almacenando: La fecha y hora de la acción, el identificador del usuario, el nombre de la acción ejecutada y los valores recibidos por la acción.

6. Agregue una acción sólo para los usuarios administradores que permita descargar el registro de operaciones de todos los usuarios desarrollado en el punto anterior.