Trabajo Práctico 3

Índice

Índice 1: Uso de Funciones de Otro Archivo (Exportación e Importación)

Objetivo: Usar funciones exportadas desde otro archivo para leer un archivo JSON y devolver información ordenada.

Teoría:
La modularización del código mediante la exportación e importación de funciones permite una mejor organización y mantenimiento del código. En Node.js, puedes exportar funciones, objetos o clases desde un archivo utilizando `export` y luego importarlos en otros archivos usando `import`.

Pasos y Ejemplo:

1. **Crear el archivo `utils.mjs` con la función exportada:**

   Contenido de `utils.mjs`:

```
import fs from 'fs';

// Clase para representar un Superhéroe
class Superheroe {
        constructor(id, nombreSuperheroe, nombreReal, nombreSociedad, edad, planetaO
        this.id = id;
        this.nombreSuperheroe = nombreSuperheroe;
        this.nombreReal = nombreReal;
        this.nombreSociedad = nombreSociedad;
        this.edad = edad;
        this.planetaOrigen = planetaOrigen;
        this.debilidad = debilidad;
        this.poder = poder;
```

```
        this.habilidadEspecial = habilidadEspecial;
        this.aliado = aliado;
        this.enemigo = enemigo;
        }
}

// Función para leer y ordenar los superhéroes
export function leerSuperheroes(ruta) {
        const datos = fs.readFileSync(ruta, 'utf8');
        const superheroesArray = JSON.parse(datos);

        // Convertir a instancias de Superheroe
        const superheroes = superheroesArray.map(
        hero => new Superheroe(hero.id, hero.nombreSuperheroe, hero.nombreReal, hero
        );

        // Ordenar por nombre de superhéroe
        superheroes.sort((a, b) => a.nombreSuperheroe.localeCompare(b.nombreSuperhero
        return superheroes;
}
```

Explicación:

- **Clase `Superheroe`**: Define un modelo para los superhéroes con todas las propiedades necesarias.
- **Función `leerSuperheroes`**: Lee el archivo JSON, lo convierte en objetos `Superheroe`, los ordena por el nombre del superhéroe y devuelve la lista ordenada.

2. **Crear el archivo `index.mjs` para utilizar la función exportada:**

Contenido de `index.mjs`:

```
import { leerSuperheroes } from './utils.mjs';

// Leer y mostrar la lista de superhéroes ordenada
const superheroes = leerSuperheroes('./superheroes.txt');
console.log('Superhéroes ordenados:');
console.log(superheroes);
```

Explicación:

- **Importación de la función**: Se importa la función `leerSuperheroes` desde `utils.mjs`.
- **Uso de la función**: Se llama a la función para leer y ordenar los superhéroes desde el archivo `superheroes.txt`, y se imprime la lista ordenada en la consola.

3. **Ejemplo de archivo `superheroes.txt`:**

Contenido de `superheroes.txt`:

```
[
        {"id": 1, "nombreSuperheroe": "Spiderman", "nombreReal": "Peter Parker", "nor
        {"id": 2, "nombreSuperheroe": "Ironman", "nombreReal": "Tony Stark", "nombre
        {"id": 3, "nombreSuperheroe": "Thor", "nombreReal": "Thor Odinson", "nombreS
]
```

Explicación:

- **Contenido del archivo**: El archivo JSON contiene una lista de superhéroes con todas las propiedades necesarias para el ejemplo.

Práctica: Modificación de la Lista de Superhéroes

Objetivo: Adaptar el archivo `index.mjs` y `util.mjs` para agregar nuevos superhéroes a la lista existente utilizando la función `agregarSuperheroes` de `utils.mjs`.

Links:

- `util.mjs`
- `index.mjs`
- superheroes.txt
- agregarSuperheroes.txt

Archivos Proporcionados:

1. `agregarSuperheroes.txt` - Cada integrante del grupo debe inventar un superhéroe y agregarlo a esta lista Crear, editar
2. `util.mjs` - Hay que crear una funcion agregarSuperheroes(archOriginal,archNuevo) y exportarla editar
3. `index.mjs` - Hay que llamar a funcion agregarSuperheroes de utils editar
4. `superheroes.txt` - Contiene la lista actual de superhéroes en formato JSON. ok

Contenido de `superheroes.txt`:

```
[
    {"id": 1, "nombreSuperheroe": "Spiderman", "nombreReal": "Peter Parker", "nombreSocie
    {"id": 2, "nombreSuperheroe": "Ironman", "nombreReal": "Tony Stark", "nombreSociedad"
    {"id": 3, "nombreSuperheroe": "Thor", "nombreReal": "Thor Odinson", "nombreSociedad":
]
```

Contenido de `agregarSuperheroes.txt` (nombres elegidos al azar):

```
[
    {"id": 4, "nombreSuperheroe": "Black Widow", "nombreReal": "Natasha Romanoff", "nombr
    {"id": 5, "nombreSuperheroe": "Hulk", "nombreReal": "Bruce Banner", "nombreSociedad":
    {"id": 6, "nombreSuperheroe": "Captain America", "nombreReal": "Steve Rogers", "nombr
    {"id": 7, "nombreSuperheroe": "Doctor Strange", "nombreReal": "Stephen Strange", "nom
]
```

Tareas:

1. **Modificar `util.mjs`:**

   Agregar la Función `agregarSuperheroes`: Esta función leerá los nuevos superhéroes desde un archivo y los añadirá a la lista existente, luego guardará la lista actualizada en el archivo original.

   Actualizar `utils.mjs`: Aquí te muestro cómo puedes implementar la función `agregarSuperheroes`.

```
import fs from 'fs';

// Clase para representar un Superhéroe
class Superheroe {
        constructor(id, nombreSuperheroe, nombreReal, nombreSociedad, edad, planetaO
        this.id = id;
        this.nombreSuperheroe = nombreSuperheroe;
        this.nombreReal = nombreReal;
        this.nombreSociedad = nombreSociedad;
        this.edad = edad;
        this.planetaOrigen = planetaOrigen;
        this.debilidad = debilidad;
        this.poder = poder;
        this.habilidadEspecial = habilidadEspecial;
        this.aliado = aliado;
        this.enemigo = enemigo;
        }
}

// Función para leer y ordenar los superhéroes
export function leerSuperheroes(ruta) {
        const datos = fs.readFileSync(ruta, 'utf8');
        const superheroesArray = JSON.parse(datos);

        // Convertir a instancias de Superheroe
        const superheroes = superheroesArray.map(
        hero => new Superheroe(hero.id, hero.nombreSuperheroe, hero.nombreReal, hero
        );
```

```
        // Ordenar por nombre de superhéroe
        superheroes.sort((a, b) => a.nombreSuperheroe.localeCompare(b.nombreSuperhero
        return superheroes;
}

// Nueva función para agregar superhéroes
export function agregarSuperheroes(rutaOriginal, rutaNuevos) {
        const datosOriginales = fs.readFileSync(rutaOriginal, 'utf8');
        const datosNuevos = fs.readFileSync(rutaNuevos, 'utf8');

        const superheroesOriginales = JSON.parse(datosOriginales);
        const nuevosSuperheroes = JSON.parse(datosNuevos);

        // Convertir los nuevos superhéroes a instancias de Superheroe
        const instanciasNuevos = nuevosSuperheroes.map(
                hero => new Superheroe(hero.id, hero.nombreSuperheroe, hero.nombreRea
        );

        // Combinar listas
        const listaActualizada = [...superheroesOriginales, ...instanciasNuevos];

        // Guardar la lista actualizada
        fs.writeFileSync(rutaOriginal, JSON.stringify(listaActualizada, null, 2), 'ut
        console.log('Lista de superhéroes actualizada con éxito.');
}
```

Explicación:

1. **Lectura de Archivos:**
   - `fs.readFileSync(rutaOriginal, 'utf8')` lee el archivo de superhéroes existente.
   - `fs.readFileSync(rutaNuevos, 'utf8')` lee el archivo con los nuevos superhéroes.
2. **Conversión a Instancias de `Superheroe`:**
   - La función `agregarSuperheroes` convierte los datos de los nuevos superhéroes en instancias de `Superheroe`.
3. **Combinar Listas y Guardar:**
   - Se combinan las listas de superhéroes originales y nuevos.
   - Se guarda la lista combinada de vuelta en el archivo original.
4. **Mensaje de Éxito:**
   - Se imprime un mensaje en la consola para confirmar que la actualización fue exitosa.

Este enfoque mantiene la estructura y formato de los datos y asegura que los nuevos superhéroes se añadan correctamente a la lista existente.

2. **Modificar `index.mjs` :**

   Debes actualizar el archivo `index.mjs` para utilizar la nueva función `agregarSuperheroes` en lugar de solo leer y mostrar la lista de superhéroes.

   Código de `index.mjs` Modificado:

   Aquí está el código actualizado para `index.mjs` :

```
import { leerSuperheroes, agregarSuperheroes } from './utils.mjs';

    const archivoOriginal = './superheroes.txt';
    const archivoNuevos = './agregarSuperheroes.txt';

    // Agregar nuevos superhéroes
    agregarSuperheroes(archivoOriginal, archivoNuevos);

    // Leer y mostrar la lista actualizada de superhéroes ordenada
    const superheroes = leerSuperheroes(archivoOriginal);
    console.log('Superhéroes ordenados:');
    console.log(superheroes);
```

   Explicación del Código:

   - **Importar Funciones:** Se importan las funciones `leerSuperheroes` y `agregarSuperheroes` del archivo `utils.mjs` .
   - **Agregar Nuevos Superhéroes:** Se llama a `agregarSuperheroes` para combinar los nuevos superhéroes con la lista existente.
   - **Leer y Mostrar Lista Actualizada:** Después de añadir los nuevos superhéroes, se lee la lista actualizada y se muestra en la consola.

3. **Instrucciones Adicionales:**

   1. **Ejecutar la Modificación:**
      Asegúrate de que los archivos `superheroes.txt` y `agregarSuperheroes.txt` estén en la misma carpeta que `index.mjs` y `utils.mjs` .

   2. **Verificar Resultados:**
      Después de ejecutar `index.mjs` , verifica que la lista de superhéroes en `superheroes.txt` se ha actualizado correctamente y muestra todos los superhéroes ordenados.

Links:

- `util.mjs`
- `index.mjs`
- superheroes.txt

- `util final.mjs`
- `index final.mjs`
- agregarSuperheroes.txt

Esta práctica esta diseñada para ayudarte a entender cómo actualizar y combinar datos en archivos JSON usando Node.js. Es importante que intentes resolverla y consultes cuando tengas dudas. ¡Buena suerte!

---

2. Gestión de Módulos con NPM

2.1. Introducción a NPM

- **Descripción General**: NPM (Node Package Manager) es el gestor de paquetes predeterminado para Node.js, permitiendo a los desarrolladores instalar, actualizar y gestionar módulos y bibliotecas en sus proyectos.
- **Beneficios**: Facilita la integración de dependencias, gestión de versiones y automatización de tareas a través de scripts. Comparado con otros gestores como Yarn, NPM ofrece una amplia base de paquetes y una comunidad activa.
- **Teoría**: NPM simplifica la reutilización de código al proporcionar un repositorio centralizado de módulos, y ayuda a mantener un entorno de desarrollo organizado y consistente.

2.2. Instalación y Configuración de NPM

- **Instalación**: NPM se instala junto con Node.js. El proceso de instalación varía según el sistema operativo (Windows, macOS, Linux). Se puede verificar la instalación con `npm --version` y `node --version`.
- **Configuración Inicial**: Crear un archivo `package.json` con `npm init`. Este archivo es crucial para gestionar las dependencias y scripts del proyecto.
- **Comandos Básicos**:
  - `npm init`: Inicializa un nuevo proyecto y genera un archivo `package.json`.
  - `npm install`: Instala paquetes y sus dependencias, generando un archivo `package-lock.json` para un control de versiones exacto.

- **Teoría**: El archivo `package.json` contiene metadatos del proyecto, dependencias, y scripts personalizados. La correcta configuración de este archivo asegura que el entorno de desarrollo y producción esté alineado.

2.3. Manejo de Dependencias

- **Tipos de Dependencias**:
  - **Dependencias de Producción**: Paquetes necesarios para ejecutar la aplicación en un entorno en vivo (ej. `express`).
  - **Dependencias de Desarrollo**: Paquetes útiles solo durante el desarrollo, como herramientas de prueba y linters (ej. `jest`, `eslint`).
- **Comandos para Manejo de Dependencias**:
  - `npm install <paquete>`: Instala una dependencia y la añade al archivo `package.json`.
  - `npm uninstall <paquete>`: Elimina una dependencia del proyecto.
  - `npm update`: Actualiza las dependencias a versiones compatibles con las restricciones especificadas en `package.json`.
  - **Versionado**: Especifica rangos de versión en `package.json` para controlar la compatibilidad y estabilidad de las dependencias.
- **Teoría**: Comprender los tipos de dependencias y cómo gestionar sus versiones es crucial para evitar conflictos y asegurar la estabilidad del proyecto.

2.4. Scripts en `package.json`

- **Definición de Scripts**: Los scripts en `package.json` automatizan tareas como pruebas y despliegue. Definir scripts personalizados mejora la eficiencia y consistencia en el desarrollo.
- **Comandos Comunes**:
  - `npm start`: Ejecuta el script definido para iniciar la aplicación, comúnmente configurado para iniciar el servidor.
  - `npm test`: Ejecuta el script de pruebas definido para el proyecto.
- **Ejemplos de Scripts**:
  - Configuración de scripts para tareas recurrentes, como compilación de código, minificación de archivos, y ejecución de tareas de linting.
- **Teoría**: Utilizar scripts en `package.json` estandariza procesos y facilita la automatización, reduciendo errores manuales y agilizando el flujo de trabajo.

2.5. Trabajo con Paquetes Públicos y Privados

- **Paquetes Públicos**: Se pueden buscar e instalar paquetes desde el registro público de NPM utilizando `npm search` y `npm install`.
- **Paquetes Privados**: Configuración de un registro privado para gestionar paquetes que no deben ser compartidos públicamente. Se utiliza el comando `npm publish` para publicar paquetes privados.
- **Autenticación**: Gestión de credenciales para acceder a paquetes privados, configurando el archivo `.npmrc`.

- **Teoría**: Trabajar con paquetes privados y públicos permite una flexibilidad en la gestión de dependencias, adaptándose a diferentes necesidades de seguridad y colaboración.

2.6. Prácticas y Herramientas de NPM

- **NPM Audit**: Usa el comando `npm audit` para identificar vulnerabilidades en las dependencias del proyecto. Esto ayuda a mantener la seguridad del proyecto al detectar y resolver problemas de seguridad.
- **NPM outdated**: Verifica las versiones desactualizadas de las dependencias con `npm outdated`, facilitando la actualización proactiva.
- **NPM ci**: El comando `npm ci` instala dependencias de manera limpia y reproducible basándose en el archivo `package-lock.json`, útil para entornos de integración continua.
- **Teoría**: Las herramientas de NPM ayudan a mantener un entorno de desarrollo seguro y actualizado, y facilitan la integración continua y la gestión de versiones.

Este índice proporciona una guía integral para gestionar módulos y dependencias en un proyecto Node.js usando NPM.

Índice 3: Introducción a Express

Objetivo: Introducir y configurar el framework Express para el desarrollo de aplicaciones web y APIs en Node.js, explorar diferentes tipos de ruteo y realizar actividades prácticas para manejar parámetros en solicitudes `GET`.

Teoría:
Express es un framework de servidor web para Node.js que proporciona una infraestructura ligera y flexible para el desarrollo de aplicaciones web y APIs. Su diseño modular y su enfoque en middleware permiten una gran facilidad para gestionar el flujo de solicitudes y respuestas, hacer rutas condicionales, y manejar la lógica de la aplicación de manera eficiente.

- **Conceptos Clave**:
  - **Middleware**: Las funciones de middleware en Express permiten interceptar y modificar las solicitudes y respuestas en diferentes puntos del ciclo de vida de una solicitud HTTP. Los middleware son útiles para tareas como logging, autenticación, y manejo de errores. Se pueden aplicar a nivel global para todas las rutas o a nivel específico para rutas individuales.
  - **Ruteo**: Express permite definir rutas de manera declarativa. Cada ruta se asocia con una función que maneja las solicitudes para esa ruta específica. Los métodos HTTP (GET, POST, PUT, DELETE) se utilizan para definir cómo se manejan diferentes tipos de solicitudes en cada ruta.

- **Gestión de Solicitudes y Respuestas**: Express simplifica la gestión de solicitudes y respuestas HTTP, proporcionando métodos para enviar respuestas de diferentes tipos (texto, JSON, HTML) y para acceder a los datos enviados en la solicitud.

Pasos y Ejemplo:

1. **Instalación de Express**:
   Instala Express en tu proyecto usando NPM para añadir el paquete a tus dependencias.

   Comando para instalar Express:

```
npm install express
```

2. **Configuración Básica del Servidor**:
   Configura un servidor básico en Express creando un archivo `app.js` y definiendo una ruta simple.

   Contenido de `app.js`:

```
import express from 'express';

// Crear una instancia de Express
const app = express();

// Configurar el puerto en el que el servidor escuchará
const PORT = 3000;

// Ruta básica
app.get('/', (req, res) => {
        res.send('¡Hola, mundo!');
});

// Iniciar el servidor
app.listen(PORT, () => {
        console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

3. **Ejemplos de Ruteo en Express**:

   Ruteo Básico:
   Define rutas que responden a solicitudes `GET` específicas.

```
// Ruta GET para el home
                // Solicitud: http://localhost:3000/
                app.get('/', (req, res) => {
                        res.send('Página de inicio');
                });
```

```
// Ruta GET para recibir datos simples
// Solicitud: http://localhost:3000/data
app.get('/data', (req, res) => {
        res.send('Datos recibidos');
});
```

Ruteo con Parámetros:

Maneja rutas que contienen parámetros dinámicos en la URL.

```
// Ruta GET con parámetro de ruta
// Solicitud: http://localhost:3000/user/123
app.get('/user/:id', (req, res) => {
        const userId = req.params.id;
        res.send(`Perfil del usuario con ID: ${userId}`);
});

// Ruta GET con múltiples parámetros
// Solicitud: http://localhost:3000/product/electronics/456
app.get('/product/:category/:id', (req, res) => {
        const { category, id } = req.params;
        res.send(`Categoría: ${category}, ID del producto: ${id}`);
});
```

Ruteo con Consultas:

Maneja rutas que utilizan parámetros de consulta en la URL.

```
// Ruta GET con parámetro de consulta
// Solicitud: http://localhost:3000/search?q=javascript
app.get('/search', (req, res) => {
        const query = req.query.q;
        res.send(`Resultados de búsqueda para: ${query}`);
});

// Ruta GET con múltiples parámetros de consulta
// Solicitud: http://localhost:3000/filter?type=book&minPrice=10&maxPrice=50
app.get('/filter', (req, res) => {
        const { type, minPrice, maxPrice } = req.query;
        res.send(`Filtrar por tipo: ${type}, rango de precios: ${minPrice} - ${maxPr
});
```

Explicación:

- **Ruteo Básico**: Define cómo manejar diferentes tipos de solicitudes `GET` en distintas rutas. `app.get()` se usa para manejar solicitudes `GET`.

- **Ruteo con Parámetros**: Permite capturar partes de la URL que son variables y usarlas en el manejo de la solicitud. Esto es útil para crear rutas dinámicas que responden a diferentes valores.
- **Ruteo con Consultas**: Utiliza parámetros de consulta en la URL para realizar operaciones como filtrado y búsqueda. Estos parámetros se pueden acceder a través de `req.query`.

Actividades Prácticas

Actividad 1: Manejo de Parámetros de Ruta

1. **Objetivo**: Configurar un servidor Express que reciba una solicitud `GET` con un parámetro `id` en la URL y muestre ese parámetro en la consola.

   Links:

   - `serverRuta.mjs`

2. **Pasos**:

   1. Crea un archivo `server.mjs` (o usa `app.mjs` si lo prefieres).
   2. Configura una ruta que capture el parámetro `id` en la URL.
   3. Muestra el valor del parámetro `id` en la consola cuando se reciba una solicitud.

   Código:

```
import express from 'express';

const app = express();
const PORT = 3000;

// Ruta GET con parámetro de ruta
// Solicitud: http://localhost:3000/user/123
app.get('/user/:id', (req, res) => {
        const userId = req.params.id;
        console.log(`ID del usuario recibido: ${userId}`);
        res.send(`Perfil del usuario con ID: ${userId}`);
});

app.listen(PORT, () => {
        console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

Actividad 2: Manejo de Parámetros de Consulta

1. **Objetivo**: Configurar un servidor Express que reciba una solicitud `GET` con un parámetro de consulta `edad` en la URL y muestre ese parámetro en la consola.

Links:

- serverConsulta.mjs

2. **Pasos**:

    1. Crea un archivo `server.js` (o usa `app.js` si lo prefieres).
    2. Configura una ruta que capture el parámetro de consulta `edad`.
    3. Muestra el valor del parámetro `edad` en la consola cuando se reciba una solicitud.

    Código:

```
import express from 'express';

const app = express();
const PORT = 3000;

// Ruta GET con parámetro de consulta
// Solicitud: http://localhost:3000/profile?edad=30
app.get('/profile', (req, res) => {
        const edad = req.query.edad;
        console.log(`Edad recibida: ${edad}`);
        res.send(`Edad del perfil: ${edad}`);
});

app.listen(PORT, () => {
        console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

Estas actividades te permitirán practicar la configuración de rutas en Express y la gestión de parámetros de ruta y de consulta, mostrando cómo el servidor maneja estos parámetros y los presenta en la consola.
