Listas de ejercicios: Javascript asíncronico

Objetivo: La actividad tiene como objetivo evaluar tanto la habilidad técnica de los estudiantes al resolver ejercicios de programación asincrónica en JavaScript como su capacidad de explicar los conceptos vistos en clase. Se valorará no solo si resuelven al menos 5 ejercicios de los propuestos, sino también la claridad con la que presentan y justifican sus decisiones.

Instrucciones:

1. Cada estudiante debe elegir 5 ejercicios de la lista proporcionada y desarrollarlos. No copiar y hacer uso de herramientas de generación de código.

2. Durante la exposición, el estudiante debe demostrar el funcionamiento del código en vivo, explicando cómo cada pieza del código implementa conceptos de asincronía en JavaScript.

3. La exposición debe centrarse en cómo el código resuelve el problema, las decisiones de diseño, y la gestión de errores.

4. Los estudiantes también deben responder a las preguntas de análisis asociadas a cada ejercicio.

5. Formato de Exposición:

- Presentación del problema: Explicar brevemente el ejercicio elegido.

- Ejecución del código: Mostrar cómo el código funciona y cumple los requisitos del ejercicio.

- Explicación técnica: Describir los conceptos clave utilizados, como callbacks, promesas, async/await, manejo de errores, concurrencia, etc.

- Preguntas de análisis: Responder a las preguntas que acompañan cada ejercicio para mostrar una comprensión profunda de los conceptos.

6. El estudiante deberá solicitar al profesor instructor la fecha de presentación y guardar su trabajo en su repositorio personal del curso. Fecha máxima una semana antes del exámen parcial.

Ejemplo de Estructura de presentación:

1. Ejercicio 1: Callbacks y el "Callback Hell"

- Descripción: Desarrollé una aplicación que realiza varias operaciones dependientes utilizando solo callbacks, lo que provoca el infame "callback hell".

- Ejecución: (Mostrar el código ejecutándose y los resultados).

- Explicación: Aquí he utilizado callbacks para manejar la secuencia de operaciones, comenzando con la lectura de un archivo JSON. Luego, realizo una solicitud HTTP para obtener datos adicionales

de cada usuario y, finalmente, escribo los datos en un archivo nuevo. El código demuestra cómo los callbacks crean una estructura de difícil mantenimiento y cómo esto podría resolverse con promesas.

- Preguntas de análisis: Para mejorar la legibilidad del código, podría reestructurarlo usando promesas o async/await, lo que eliminaría la pirámide de callbacks. El mayor problema que encontré es la falta de control sobre los errores, lo que requiere más lógica adicional para manejar casos de error en cada nivel de callback.

2. Ejercicio 2: Encadenamiento de promesas

- Descripción: He convertido el ejercicio anterior, usando promesas para evitar el "callback hell" y hacer el código más legible.

- Ejecución: (Mostrar el código ejecutándose y los resultados).

- Explicación: Aquí utilicé fs.promises para manejar la lectura/escritura de archivos y encadené las promesas para evitar el callback hell. El flujo es más claro y los errores se manejan de forma más eficiente gracias al .catch() que centraliza la gestión de errores.

- Preguntas de análisis: Comparado con los callbacks, el uso de promesas hace que el código sea más modular y fácil de leer. Las promesas también proporcionan un mejor manejo de errores al permitir centralizar la lógica de captura.

Criterios de evaluación:

- Resolución técnica (40%): La solución presentada debe funcionar correctamente y cumplir los requisitos del ejercicio.

- Explicación de conceptos (40%): Los estudiantes deben demostrar un buen dominio de los conceptos clave (callbacks, promesas, async/await, manejo de errores, etc.).

- Claridad en la exposición (20%): La presentación debe ser clara, bien estructurada y accesible para todos los miembros del grupo.

Recomendaciones para la exposición:

- Prepara ejemplos claros que demuestren cómo las diferentes técnicas (callbacks, promesas, async/await) afectan la estructura del código.

- Comparte tus ideas sobre cómo manejar errores de forma eficiente y cómo mejorar el rendimiento del código asincrónico.

- Recuerda que la forma en que explicas y justificas tus decisiones es tan importante como el funcionamiento del código en sí.

Lista de ejercicios para resolver:

Ejercicio 1: Callbacks y el "Callback Hell"

Crea una aplicación en Node.js que realice una serie de operaciones dependientes unas de otras utilizando solo callbacks. Debe incluir:

1. Leer un archivo JSON que contiene información de usuarios.
2. Hacer una solicitud HTTP a una API externa para obtener información adicional de cada usuario.
3. Guardar los datos combinados en un nuevo archivo.

Requisito: La secuencia de operaciones debe ejecutarse solo usando callbacks, y debes enfrentarte al problema conocido como "callback hell". Evalúa qué tan fácil es mantener y escalar este código.

Preguntas de análisis:

- ¿Cómo podrías mejorar la legibilidad del código?
- ¿Qué problemas encontraste en la gestión de errores?

Ejercicio 2: Encadenamiento de Promesas

Convierte el ejercicio anterior, ahora usando promesas en lugar de callbacks. Deberás realizar las mismas operaciones (leer archivo, hacer una solicitud HTTP y guardar datos), pero usando la API de promesas.

Requisito: Usa fs.promises para manejar las operaciones de lectura/escritura de archivos y fetch o un cliente HTTP que soporte promesas.

Preguntas de análisis:

- Compara la legibilidad del código en comparación con el uso de callbacks.
- ¿Cómo maneja las promesas los errores en comparación con los callbacks?

Ejercicio 3: Encadenamiento complejo con promesas

Desarrolla un sistema de gestión de inventario de productos que, al recibir una solicitud de actualización, debe:

1. Validar los datos del producto utilizando una API de validación externa.
2. Actualizar la base de datos con la información del producto.
3. Notificar a los usuarios interesados sobre la actualización mediante una API de notificación externa.

Haz que estas operaciones estén encadenadas mediante promesas y asegúrate de que, si cualquiera de las operaciones falla, las operaciones subsecuentes no se ejecuten.

Requisito: Usa encadenamiento de promesas para estructurar las operaciones y manejar los errores de forma centralizada con .catch().

Preguntas de análisis:

- ¿Cómo gestionas el flujo de operaciones cuando una de las promesas falla?
- ¿Qué ventajas aporta el encadenamiento de promesas en comparación con el "callback hell"?

Ejercicio 4: Migración a async/await

Convierte el sistema de gestión de inventario del ejercicio anterior a usar async/await. Las operaciones deben ser las mismas (validación, actualización de base de datos, notificación).

Requisito: Usa async/await y asegúrate de que todos los errores se gestionen adecuadamente mediante try...catch.

Preguntas de análisis:

- ¿Cómo ha cambiado la legibilidad del código respecto a las promesas encadenadas?
- ¿Es más fácil manejar los errores con async/await en comparación con las promesas?

Ejercicio 5: Manejo de errores con async/await

Imagina un servicio web que consume múltiples APIs externas de datos meteorológicos. Tu tarea es implementar una función que haga solicitudes a tres servicios meteorológicos diferentes y devuelva la temperatura media de los tres. Si una de las APIs falla o no responde, la función debe seguir funcionando con las otras dos.

Requisito: Implementa la función utilizando async/await y Promise.allSettled() para asegurarte de que, si una solicitud falla, las otras continuarán.

Preguntas de análisis:

- ¿Por qué es útil Promise.allSettled() en este caso?
- ¿Cómo manejarías las respuestas fallidas dentro de tu código?

Ejercicio 6: Comparación de patrones asincrónicos

Desarrolla una aplicación de registro de usuarios en un sistema web. Debe realizar lo siguiente:

1. Validar el formulario del usuario.
2. Almacenar los datos del usuario en la base de datos.
3. Enviar un email de bienvenida.

Implementa esta aplicación usando tres enfoques distintos:

4. Usando solo callbacks.
5. Usando promesas encadenadas.
6. Usando async/await.

Requisito: Evalúa cada implementación en términos de legibilidad, manejo de errores y eficiencia.

Preguntas de análisis:

- ¿Cuál de los tres enfoques prefieres y por qué?
- ¿En qué situaciones seguirías utilizando callbacks o promesas en lugar de async/await?

Ejercicio 7: Paralelismo con Promise.all()

Desarrolla una API REST que obtenga datos de tres servicios diferentes (como servicios de clima, noticias y cotizaciones de criptomonedas) y los devuelva en una única respuesta consolidada. Los tres servicios deben consultarse en paralelo para optimizar el tiempo de respuesta.

Requisito: Usa Promise.all() para realizar las solicitudes a los tres servicios en paralelo y manejar el resultado de forma eficiente.

Preguntas de análisis:

- ¿Qué beneficios aporta Promise.all() en cuanto al rendimiento comparado con hacer las solicitudes en serie?
- ¿Cómo manejarías los errores si uno de los servicios falla?

Ejercicio 8: Cancelación de promesas

Implementa una función en Node.js que permita leer un archivo grande de forma asincrónica y que soporte la opción de cancelar la lectura si el usuario lo decide antes de que termine.

Requisito: Usa Promise y simula una funcionalidad de cancelación (no nativa en JavaScript) mediante el uso de un mecanismo externo que permita abortar la operación.

Preguntas de análisis:

- ¿Qué técnicas de manejo de promesas puedes usar para simular la cancelación?
- ¿Qué ventajas e inconvenientes tiene este enfoque en comparación con APIs de lenguajes que soportan cancelación nativa?

Ejercicio 9: Manejo avanzado de errores

Crea un sistema de autenticación que:

1. Valide las credenciales de un usuario contra una base de datos.
2. Genere un token JWT si la autenticación es exitosa.
3. Registre la actividad de inicio de sesión en un sistema externo.

Requisito: Implementa este sistema utilizando promesas y asegúrate de que si la generación del token o el registro de actividad falla, el usuario reciba una respuesta adecuada y los errores se manejen de manera centralizada.

Preguntas de análisis:

- ¿Cómo manejas los errores en cada paso del flujo sin que el sistema se detenga por completo?
- ¿Es preferible manejar todos los errores en un solo lugar o en cada promesa?

Ejercicio 10: async/await con Retries

Imagina un servicio que realiza solicitudes a una API externa, pero esta API es poco confiable y falla intermitentemente. Tu tarea es implementar una función que realice hasta tres intentos de solicitar datos de la API antes de devolver un error.

Requisito: Usa async/await y crea una lógica de reintentos que permita hacer tres intentos antes de lanzar una excepción.

Preguntas de análisis:

- ¿Cómo implementas de manera efectiva los reintentos en tu función asincrónica?
- ¿Qué pasa si la API sigue fallando después de tres intentos? ¿Cómo lo manejarías?

Ejercicio 11: Promesas y estructura de árbol

Crea una estructura de árbol que represente una jerarquía de tareas, donde cada nodo del árbol es una función asincrónica que puede tener múltiples hijos (tareas dependientes). El nodo padre solo debe ejecutarse después de que todas las promesas de sus hijos se hayan resuelto.

Requisito: Implementa la jerarquía de tareas usando promesas encadenadas o async/await, asegurando que se respete la dependencia de los hijos.

Preguntas de análisis:

- ¿Cómo puedes optimizar el código para evitar que las tareas dependientes se bloqueen innecesariamente?
- ¿Cómo manejarías errores que ocurran en uno de los nodos?

Ejercicio 12: Cacheo de resultados asincrónicos

Implementa una función en Node.js que obtenga datos de una API y los almacene en un caché (usando Redis o memoria local). Si los datos ya están en el caché, la función debe devolver los datos del caché en lugar de hacer una nueva solicitud a la API.

Requisito: Usa async/await para implementar esta lógica, asegurándote de que el caché se actualice correctamente cada cierto tiempo.

Preguntas de análisis:

- ¿Cómo manejas el vencimiento del caché y su actualización de manera eficiente?
- ¿Qué sucede si la API externa tarda mucho tiempo en responder o falla?

Ejercicio 13: Sincronización de múltiples operaciones asincrónicas

En un entorno de e-commerce, cuando un usuario realiza una compra, debes:

1. Registrar la compra en la base de datos.
2. Restar el stock disponible de los productos adquiridos.
3. Enviar un correo electrónico de confirmación al usuario.

Todas estas operaciones deben completarse con éxito para confirmar la transacción.

Requisito: Implementa este flujo de trabajo usando Promise.all() para garantizar que todas las operaciones se realicen de manera paralela. Si alguna operación falla, la transacción debe ser revertida (rollback).

Preguntas de análisis:

- ¿Cómo gestionas el rollback si una de las promesas falla?
- ¿Cómo manejarías la situación en la que un proceso como el envío de correo falla, pero los otros pasos se completan con éxito?

Ejercicio 14: Control de concurrencia

Desarrolla una función que permita descargar varios archivos grandes de una API al mismo tiempo. Sin embargo, debido a las limitaciones de la API, solo puedes hacer un máximo de 5 descargas simultáneamente.

Requisito: Usa async/await y un sistema de cola para limitar el número de solicitudes concurrentes que se pueden hacer a la API.

Preguntas de análisis:

- ¿Cómo implementas un sistema de control de concurrencia eficiente en este caso?
- ¿Cómo manejarías las solicitudes que exceden el límite de concurrencia sin perder ninguna?

Ejercicio 15: Ejecución condicional asincrónica

Crea una aplicación que se conecte a varias APIs de terceros (por ejemplo, para obtener datos financieros o información meteorológica), pero solo realice la solicitud si ciertos datos almacenados en caché han expirado o no existen.

Requisito: Usa async/await para manejar las operaciones condicionales. Si los datos no están en caché o están expirados, la función debe realizar la solicitud; de lo contrario, debe devolver los datos almacenados.

Preguntas de análisis:

- ¿Cómo decides si los datos almacenados están expirados de manera eficiente?
- ¿Cómo puedes mejorar el rendimiento de la aplicación al reducir la cantidad de solicitudes innecesarias?