ASINCRONISMO EN JAVASCRIPT

TEMA 08 – PROMESAS

8.1. Introducción

En el desarrollo moderno con JavaScript, especialmente en el entorno del navegador y en Node.js, uno de los retos más habituales es la gestión de operaciones asíncronas. Anteriormente, se utilizaban callbacks para manejar estas operaciones, lo cual podía derivar en un “callback hell” o XHR con una estructura sintáctica con cierta complejidad. Con la introducción de las *Promises* a partir de 2015, se ofreció una solución más clara y manejable para escribir, organizar y razonar sobre código asíncrono.

A continuación, profundizaremos en la base teórica de las Promesas, sus estados y sus métodos fundamentales. Además, mostraremos ejemplos simples para ilustrar su uso.

8.2. Promesas

Una *Promise* es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Técnicamente, una Promesa modela un flujo asíncrono de forma que, cuando se ejecuta una operación que implica cierta latencia (lectura de archivos, consultas a bases de datos, llamadas a APIs, entre otras), podamos tratar el resultado (o el posible error) sin necesidad de bloquear el resto de la ejecución.

Definición de una Promesa

Podemos crear una nueva promesa mediante la clase nativa Promise de JavaScript, utilizando su constructor:

```
const miPromesa = new Promise((resolve, reject) => {
  // Aquí va la operación asíncrona o un bloque de código
});
```

El constructor `Promise` recibe una función "*ejecutora*" que, a su vez, recibe dos parámetros: `resolve` y `reject`.

- *resolve(valor)* se invoca cuando la operación asíncrona ha tenido éxito o ha finalizado de forma satisfactoria, donde *valor* es el resultado que la promesa entrega.
- *reject(error)* se invoca cuando la operación ha fallado o se produce algún tipo de error, donde *error* describe la causa del fallo.

Estados de una Promesa

Las promesas en JavaScript cuentan con tres estados principales:

1. **Pending (pendiente)**: El estado inicial, antes de que se haya cumplido o rechazado.
2. **Fulfilled (cumplida)**: Significa que la promesa ha terminado de forma satisfactoria, es decir, la función *resolve* ha sido llamada con éxito.
3. **Rejected (rechazada)**: Indica que la promesa no pudo completarse correctamente y se invocó la función *reject*.

Una vez que la promesa cambia de estado a *fulfilled* o *rejected*, se considera que está "*settled*" (resuelta, en el sentido de que ya no está pendiente). Sin embargo, debemos tener presente que, una vez *settled*, el estado no puede volver a cambiar.

Para visualizarlo de una forma más clara:

1. **Pending** → pasa a → **Fulfilled**
2. **Pending** → pasa a → **Rejected**

Después de esto, la promesa se considera establecida (settled) y ya no cambiará más.

8.2. Métodos básicos: .then, .catch y .finally

Las promesas se consumen (o gestionan) mediante métodos encadenables que nos permiten procesar el resultado o el error de forma ordenada.

.then()

El método *.then()* se encarga de procesar el resultado exitoso (estado *fulfilled*) de una promesa. La sintaxis básica es la siguiente:

```
miPromesa.then( (resultado) => { // Manejo de la respuesta, si todo salió bien
  }
);
```

El callback que se pasa a *.then()* se ejecuta únicamente cuando la promesa ha sido resuelta de manera satisfactoria (se ha llamado a *resolve*). Además, *.then()* puede devolver otra promesa, lo que permite encadenar múltiples operaciones asíncronas de manera más clara:

```
miPromesa
  .then((resultado) => {
    // primer bloque de manejo
    return otroValor; // puede ser un valor, o incluso otra promesa
  })
  .then((resultado2) => {
    // bloque de manejo del resultado devuelto por el .then anterior
  });
```

.catch()

El método *.catch()* se utiliza para capturar y manejar las situaciones en las que la promesa es rechazada (estado *rejected*). Es decir, cuando ocurre un error y se llama a la función *reject*:

```
miPromesa.catch(
  (error) => {
    // Manejo de errores
    console.error("Ha ocurrido un error:", error);
  }
);
```

En combinación con *.then()*, *.catch()* nos ofrece una forma clara de separar la lógica de éxito de la de error, incrementando la legibilidad:

```
miPromesa
  .then((resultado) => {
    // Si todo va bien
    console.log("Resultado:", resultado);
  })
  .catch((error) => {
    // Si algo falla
    console.error("Se ha producido un error:", error);
  });
```

Por otro lado, si dentro de un *.then()* ocurre un error de ejecución (por ejemplo, una excepción no controlada), este error también será

“encapsulado” en la promesa y podrá ser recogido más adelante por un *.catch()* posterior.

.finally()

El método *.finally()* se ejecuta siempre, independientemente de si la promesa ha sido cumplida o rechazada. Su principal utilidad radica en operaciones de limpieza o en pasos finales que deseamos realizar sin importar el resultado:

```
miPromesa
  .then((resultado) => {
    console.log("Resultado:", resultado);
  })
  .catch((error) => {
    console.error("Error en la promesa:", error);
  })
  .finally(() => {
    console.log("Esta parte se ejecutará siempre");
  });
```

8.3. EJEMPLOS

EJEMPLO 1: Funcionamiento de una Promise

Supongamos que deseamos simular una operación asíncrona que tarde cierto tiempo (como podría ser una petición a una API). Para ello, utilizaremos *setTimeout*, que nos permitirá ejecutar código tras un retardo determinado:

```
function simularOperacionAsincrona() {
  return new Promise((resolve, reject) => {
    console.log("Iniciando operación asíncrona...");
    setTimeout(() => {
      const exito = true; // Cambiar a false para probar el caso de error
      if (exito) {
        resolve("Operación completada con éxito.");
      } else {
        reject("Ocurrió un error durante la operación.");
```

```
      }
    }, 2000); // 2 segundos de retardo
  });
}


simularOperacionAsincrona()
  .then((mensaje) => {
    console.log("Mensaje de éxito:", mensaje);
  })
  .catch((error) => {
    console.error("Mensaje de error:", error);
  })
  .finally(() => {
    console.log("Fin de la operación asíncrona.");
  });
```

1. Se define la función *`simularOperacionAsincrona()`*, que retorna una nueva promesa.
2. En el *`setTimeout`*, se determina a través de la variable *`exito`* si la promesa se resuelve correctamente (*`resolve`*) o se rechaza (*`reject`*).
3. Al llamar a *`simularOperacionAsincrona()`*, utilizamos *`.then()`* para manejar el éxito, *`.catch()`* para manejar el error y *`.finally()`* para cualquier paso final (por ejemplo, registro en consola).

EJEMPLO 2: Encadenamiento de Promesas

En muchos casos, deberemos realizar varias operaciones asíncronas secuenciales, donde el resultado de una sirva como entrada a la siguiente. Con promesas, podemos hacerlo de manera clara y ordenada:

```
function obtenerUsuario() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const usuario = { id: 1, nombre: "Juan Pérez" };
      // Suponemos que la obtención del usuario fue exitosa
      resolve(usuario);
    }, 1000);
  });
}
```

```
function obtenerPedidosPorUsuario(usuarioId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos que se obtienen pedidos del usuario
      const pedidos = [
        { pedidoId: 101, usuarioId: usuarioId, producto: "Libro" },
        { pedidoId: 102, usuarioId: usuarioId, producto: "Camiseta" },
      ];
      resolve(pedidos);
    }, 1500);
  });
}

// Encadenamiento
obtenerUsuario()
  .then((usuario) => {
    console.log("Usuario obtenido:", usuario);
    return obtenerPedidosPorUsuario(usuario.id);
  })
  .then((pedidos) => {
    console.log("Pedidos del usuario:", pedidos);
  })
  .catch((error) => {
    console.error("Error en la cadena de promesas:", error);
  })
  .finally(() => {
    console.log("Fin de la ejecución.");
  });
```

En este ejemplo:

1. Primero obtenemos los datos de un usuario mediante la función *`obtenerUsuario()`*.
2. Con el resultado exitoso de la promesa anterior, llamamos a *`obtenerPedidosPorUsuario(usuario.id)`*.

3. Si en algún punto se produce un error (por ejemplo, si *obtenerUsuario()* o *obtenerPedidosPorUsuario()* llama a `reject`), la ejecución pasa de inmediato al método *.catch()*.
4. Finalmente, con *.finally()* cerramos el flujo con una notificación final.

EJEMPLO 3: Promesa estado fullfilled y rejected

En este apartado, mostraremos cómo se produce la transición entre los estados de *pending* (pendiente), *fulfilled* (cumplida) y *rejected* (rechazada). Para ello, crearemos promesas sencillas que, transcurrido un tiempo, se resuelven satisfactoriamente o se rechazan.

Promesa *fulfilled*

```
function promesaFulfilled() {
  return new Promise((resolve, reject) => {
    console.log("Promesa en estado PENDING...");

    // Simulamos una operación asíncrona
    setTimeout(() => {
      console.log("Operación completada con éxito.");
      // Llamamos a resolve para indicar que la promesa pasa a estado FULFILLED
      resolve("Esta promesa ha sido resuelta satisfactoriamente.");
    }, 2000);
  });
}
// Consumimos la promesa
promesaFulfilled()
  .then((resultado) => {
    console.log("Estado Fulfilled:", resultado);
  })
  .catch((error) => {
    // Este bloque no se ejecutará en este ejemplo
    console.error("Estado Rejected:", error);
  });
```

En este ejemplo:

1. Se crea la promesa y se informa que inicialmente está en *pending*.

2. Pasados 2 segundos, se llama a *resolve*, lo cual la lleva a *fulfilled*.
3. Al consumirse la promesa con *.then()*, se muestra por consola el resultado de éxito.

Promesa *rejected*

```
function promesaRejected() {
  return new Promise((resolve, reject) => {
    console.log("Promesa en estado PENDING...");
    // Simulamos una operación asíncrona
    setTimeout(() => {
      console.log("Operación fallida.");
      // Llamamos a reject para indicar que la promesa pasa a estado REJECTED
      reject("La promesa ha sido rechazada debido a un error.");
    }, 2000);
  });
}
// Consumimos la promesa
promesaRejected()
  .then((resultado) => {
    // Este bloque no se ejecutará en este ejemplo
    console.log("Estado Fulfilled:", resultado);
  })
  .catch((error) => {
    console.error("Estado Rejected:", error);
  });
```

En este caso:

1. Se inicia la promesa en estado *pending*.
2. Tras un retardo, se llama a *reject*, lo que provoca el estado *rejected*.
3. El *.then()* no se ejecuta, y el flujo pasa a *.catch()*, donde se gestionará el error.

EJEMPLO 4: Encadenamiento de promesas y majeno independiente de errores

En este ejemplo, mezclamos promesas que funcionan bien con otras que podrían fallar, y mostramos cómo *.catch()* puede aparecer en diferentes partes de la cadena, o agruparse en uno solo al final.

```
function validarToken() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const tokenValido = Math.random() > 0.3; // 70% de probabilidad de éxito
      if (tokenValido) {
        resolve("Token válido.");
      } else {
        reject("Token inválido, no autorizado.");
      }
    }, 800);
  });
}

function solicitarDatosPrivados() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos obtención de datos
      resolve({ informacion: "Datos privados del usuario." });
    }, 1200);
  });
}

function procesarDatos(datos) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos un posible error en el procesamiento
      const procesoExitoso = Math.random() > 0.5;
      if (procesoExitoso) {
        resolve(`Datos procesados con éxito: ${JSON.stringify(datos)}`);
      } else {
        reject("Hubo un problema al procesar los datos.");
      }
    }, 1000);
```

```
  });
}
// Cadena de Promesas
validarToken()
  .then((mensajeToken) => {
    console.log(mensajeToken);
    // Si la validación fue exitosa, solicitamos datos privados
    return solicitarDatosPrivados();
  })
  .then((datosPrivados) => {
    console.log("Datos privados obtenidos:", datosPrivados);
    // Encadenamos el procesamiento de datos
    return procesarDatos(datosPrivados);
  })
  .then((resultadoProcesado) => {
    console.log(resultadoProcesado);
  })
  .catch((error) => {
    // Cualquier error que ocurra en la cadena es capturado aquí
    console.error("Ha ocurrido un error en alguna parte del proceso:", error);
  })
  .finally(() => {
    console.log("Proceso de validación y procesamiento finalizado.");
  });
```

Observaciones:

1. *`validarToken()`* puede fallar si el token no es válido (rechaza la promesa).
2. *`solicitarDatosPrivados()`* no falla en este ejemplo (siempre resuelve).
3. *`procesarDatos()`* puede fallar aleatoriamente al procesar los datos (rechaza la promesa).
4. Cualquier rechazo en la cadena se intercepta en el *`.catch()`*.
5. *`.finally()`* muestra el mensaje final incondicionalmente.

8.4.Conclusiones

Las Promesas en JavaScript suponen un gran avance con respecto al modelo tradicional basado exclusivamente en *callbacks*. Su flujo de trabajo permite escribir código asíncrono legible y fácil de mantener, reduciendo el riesgo de “callback hell” y ofreciendo un método más claro para encadenar y manejar diferentes etapas de un proceso asíncrono.

Dominar los estados de las promesas y los métodos `.then()`, `.catch()` y `.finally()` es un paso imprescindible para comprender el funcionamiento de JavaScript en profundidad, especialmente en aplicaciones web modernas y en servicios backend con Node.js. A partir de estas bases, se puede construir conocimiento sólido sobre otras herramientas asíncronas, como `async/await`, que internamente también hacen uso de las promesas.

En resumen, las promesas permiten:

1. Ejecutar tareas asíncronas de forma no bloqueante.
2. Manejar adecuadamente distintos resultados (éxito o error).
3. Encadenar operaciones asíncronas con facilidad.
4. Mejorar la legibilidad y mantenimiento del código.