UT 04 - Otros aspectos de JavaScript

(I) Gestión de errores

Contenido

[Gestión de errores: "try... catch" 1](#_Toc232473915)

[La sintaxis “try … catch” 2](#_Toc832959833)

[El objeto Error 2](#_Toc502904986)

[Opcional “catch binding” 3](#_Toc124874768)

[Utilizando “try catch” 3](#_Toc1761327486)

[Arrojando nuestros propios errores 4](#_Toc536939813)

[El operador Throw 4](#_Toc2121292244)

[Re-lanzar errores 5](#_Toc1081196822)

[try … catch...finally 6](#_Toc430182812)

[Global catch 8](#_Toc1469857094)

[Resumen 9](#_Toc1979124625)

[Errores personalizados 9](#_Toc672289242)

[Extendiendo la clase Error 10](#_Toc1287844607)

[Más allá de la herencia 12](#_Toc2108144047)

[Envolver las excepciones 13](#_Toc472272166)

# Gestión de errores: "try... catch"

No importa lo buenos programadores que seamos, a veces nuestros scripts tienen errores. Puede ocurrir por nuestros fallos, una entrada de usuario inesperada, una respuesta errónea del servidor y cientos de motivos más.

Normalmente, un script “muere” (se detiene inmediatamente) en caso de error, escribiendo el error en la consola.

Pero existe una estructura sintáctica try … catch que nos permite “recoger” los errores de modo que el script pueda, en lugar de morir, hacer algo más razonable.

## La sintaxis “try … catch”

La estructura try...catch tiene dos bloques principales: try y catch:

try {

// código

} catch (err) {

// gestión del error

}

Esto funciona así:

1. Primero se ejecuta el código que está dentro del try { }.
2. Si no hay errores, el bloque catch(error) {} se ignora: se pasa a ejecutar lo que haya después del bloque try {}... catch {}
3. Si ocurre un error, la ejecución del bloque try {} se detiene, y el flujo continúa en el bloque catch(error){}. La variable err contendrá un objeto de tipo Error con detalles sobre lo que ha ocurrido.

De este modo, un error dentro del try {} no mata al script: tenemos la oportunidad de manejar el error en el catch(){}

Importante: try … catch sólo funciona con errores en tiempo de ejecución, por tanto, en un código válido. A esos errores se les llama "errores en tiempo de ejecución" o a veces "excepciones".

Importante: try … catch trabaja síncronamente. Si dentro del try{} se llama a una función asíncrona (ej. setTimeout()) y en esa función ocurre un error; try...catch no lo detectará.

Para detectarlos, habría que utilizar la estructura try...catch dentro del cuerpo de la función.

Nota: en lugar de err, podemos utilizar otro nombre de variable.

## El objeto Error

Cuando ocurre un error, JavaScript genera un objeto que contiene los detalles sobre el error. El objeto se pasa como argumento a catch.

Todos los errores tienen las dos propiedades principales:

* name: nombre del error. Por ejemplo, para una variable indefinida será "Reference Error".
* message: Texto-mensaje sobre los detalles del error.

Existen otras propiedades no estándar disponibles en la mayoría de los entornos. Una de las más ampliamente soportados es:

* stack: Pila de llamadas: una cadena con información sobre la secuencia de llamadas anidadas que llega hasta el error.

## Opcional “catch binding”

Es una adición reciente al lenguaje.

Es una variación de la sintaxis del catch, omitiendo el error(), si no se necesita.

try {

// código

} catch { // Sin (error)

// gestión del error

}

## Utilizando “try catch”

Veamos un caso real de uso de try … catch.

Como ya sabemos, JavaScript proporciona el método JSON.parse(str) para leer valores codificados como JSON.

Normalmente se utiliza para decodificar datos recibidos de la red, de un servidor o de otra fuente.

Nosotros podemos recibir la cadena y decodificarla llamando a JSON.parse:

let json = '{"name":"John", "age": 30}'; // cadena JSON  
let user = JSON.parse(json); // convierte la cadena JSON en un objeto

// Podemos acceder a las propiedades del objeto  
alert( user.name ); // John  
alert( user.age ); // 30

¿Qué ocurre si la cadena json está mal formada, es incorrecto? El método JSON.parse() produce un error y se detiene la ejecución del script. ¿Es esto lo que espera el usuario? Parece que no, porque el usuario al menos esperaría un mensaje de error. Esto se puede hacer utilizando try … catch.

let json = "{ bad json }";

try {  
 let user = JSON.parse(json); // <-- si hay un error...  
 alert( user.name ); // esto no se ejecuta

} catch (err) { // ...la ejecución sigue aquí  
 alert( "Disculpas, los datos tienen error, intentaremos pedirl los datos de nuevo." );  
 alert( err.name );  
 alert( err.message );  
}

Hemos utilizado try … catch. Para mostrar un mensaje, pero podemos hacer mucho más: enviar una nueva petición al servidor, sugerir una alternativa al usuario, etc. Todo mucho mejor que simplemente detener la ejecución.

## Arrojando nuestros propios errores

Supongamos que el json es sintácticamente correcto, pero no contiene una propiedad name que debería tener. Por ejemplo,

let json = '{ "age": 30 }';

try {  
 let user = JSON.parse(json);   
 alert( user.name ); // no name!   
} catch (err) {  
 alert( "doesn't execute" );   
}

En este caso, el JSON.parse se ejecuta correctamente, pero para nosotros es incorrecto porque debería tener la propiedad name. Podemos unificar la gestión de errores utilizando el operador throw.

### El operador throw

Este operador genera un error. La sintaxis es:

throw <error object>

Técnicamente, podemos utilizar cualquier cosa como error, incluso datos de tipo primitivo, como un número o una cadena. Pero es mejor utilizar objetos, preferiblemente con las propiedades name y message (por compatibilidad con los errores que genera JavaScript).

JavaScript proporciona constructores para algunos tipos de errores: Error, SyntaxError, ReferenceError, TypeError, etc.

Estos constructores se utilizan así:

let error = new Error(message);

let error = new SyntaxError(message);

let error = new ReferenceError(message);

Para estos “constructores de errores” la propiedad name contiene el nombre del constructor (Error, SyntaxError, ReferenceError,…) y la propiedad message, el texto que pasamos cuando llamamos al constructor.

Ejemplo:

let error = new Error("Things happen o\_O");  
alert(error.name); // Error  
alert(error.message); // Things happen o\_O

Ejemplo:

try {  
 JSON.parse("{ bad json o\_O }");  
} catch (err) {  
 alert(err.name); // SyntaxError   
 alert(err.message); // Unexpected token b in JSON at position 2  
}

En nuestro caso, podemos considerar que la ausencia de la propiedad name sería un error sintáctico, porque los usuarios deberían tener un nombre (name).

let json = '{ "age": 30 }';  
try {   
 let user = JSON.parse(json); // OK  
 if (!user.name) {  
 throw new SyntaxError("Datos incompletos: falta name"); // (\*)  
 }  
 alert( user.name );   
} catch (err) {  
 alert( "JSON Error: " + err.message );  
}

En la línea (\*) el operador throw genera un SyntaxError con el mensaje dado, del mismo modo que lo generaría JavaScript. La ejecución dentro del try se detiene inmediatamente y el control de flujo salta al catch. De este modo, en el bloque del código que sigue a catch manejamos todos los errores: los generados por JSON.parse y por otras causas.

## Re-lanzar errores

En el ejemplo anterior utilizamos try … catch para manejar datos incorrectos. Pero, ¿es posible que otro error inesperado ocurra dentro del bloque del try { }? Como un error de programación (variable no definida) o algo así, que no es un error de datos incorrectos.

En el ejemplo anterior, si olvidamos escribir “let” en la declaración de la variable user, obtendremos un mensaje de error incorrecto, como podemos ver a continuación:

let json = '{ "age": 30 }';

try {  
 user = JSON.parse(json); // <-- falta "let" antes de user   
 // ...   
} catch (err) {  
 alert("JSON Error: " + err);  
 // JSON Error: ReferenceError: user is not defined  
 // … pero en realidad no es un error JSON   
}

Es decir, el error puede ser de distintos tipos, no sólo un error JSON. Como nosotros sólo queremos "tratar" ese error, necesitaríamos hacer dos cosas:

* Poder determinar si un error es de un tipo o de otro
* Cuando no queremos (o no podemos) “tratar” un error, volver a enviarlo o lanzarlo “hacia arriba” para que lo gestione un try ... catch superior o el propio entorno, con su gestión por defecto de los errores.

Para determinar el tipo de error, podemos utilizar el operador instanceOf. Algunos tipos de errores pueden ser SyntaxError, ReferenceError...

Para lanzar “hacia arriba” el error, utlizaremos throw err dentro del catch.

El código quedaría así:

let json = '{ "age": 30 }'; // incomplete data

try {  
 let user = JSON.parse(json);  
 if (!user.name) {  
 throw new SyntaxError("Incomplete data: no name");   
 }  
 blabla(); // unexpected error  
 alert( user.name );  
} catch (err) {  
 if (err instanceof SyntaxError) { (\*)  
 alert( "JSON Error: " + err.message );   
 } else {  
 throw err; // reenvía el error (\*\*)  
 }   
}

## try … catch...finally

Atención, aún hay más! La estructura try … catch puede tener una cláusula adicional: finally. Esta se ejecuta en todos los casos:

* Después del try, si no hay errores
* Después del catch, si hay errores

Sintaxis:

try {

... intenta ejecuta el código ...   
} catch (err) {  
 ... maneja los errores...   
} finally {  
 ... código que se ejecucta siempres ...   
}

**Variables en try … catch … finally**

Si declaramos una variable con let dentro de un bloque de código, es decir, entre las llaves, esa variable será local a ese bloque, sólo será accesible dentro de ese bloque, por ejemplo dentro del bloque try { }.

Si queremos utilizar una variable en todos los bloques, tendremos que declararla antes.

**El bloque finally y la sentencia return**

El bloque finally se ejecuta siempre después de los bloques try {} y catch {}.

¿Qué ocurre si en uno de esos bloques hay un return? En ese caso, se ejecuta el código del bloque finally justo antes de ejecutar el return.

**¿Qué ocurre si se omite el bloque catch?**

Aunque sorprende, es muy útil cuando no queremos gestionar los errores, pero sí queremos que los procesos que se han iniciado se terminen, por ejemplo, cerrar ficheros...

## Global catch

La información de esta sección no forma parte del core de JavaScript.

Con lo anterior, podemos detectar un error si se produce en un fragmento de código determinado, pero, ¿podríamos detectar los errores que se producen en cualquier lugar del código?

Muchos entornos proporcionan un modo de hacerlo. Por ejemplo, Node.js tiene un método para ello: [process.on("uncaughtException")](https://nodejs.org/api/process.html#process_event_uncaughtexception) En los navegadores, podemos asignar una función manejadora de errores a window.onerror.

Sintaxis:

window.onerror = function(message, url, line, col, error) {  
 // ...   
};

La función del manejador de errores no es recuperar la ejecución del script (seguramente no será posible) sino guardar un log de los errores para que el desarrollador sepa lo que ha ocurrido. Hay web-services que nos proporcionan error-logging para estos casos, como [https://errorception.com](https://errorexceptcion.com) o <http://muscula.com.>

El funcionamiento suele ser:

* Registrarnos en el servicio
* Copiar el bloque de código que tenemos que insertar en el windows.onerror. Insertarlo.
* Cuando ocurra un error, se hará una petición a una web para que registre el error.
* Como desarrolladores, podemos acceder a la información sobre los errores que se han producido.

# Errores personalizados

Cuando se desarrollan aplicaciones, nos puede interesar tener nuestras propias clases de error.

Nuestros errores deberían soportar las propiedades de los errores genéricos, como message, name y, si es posible, stack. Pero podrían tener también propiedades específicas.

JavaScript permite utilizar throw sin argumento, por lo tanto, técnicamente, no es necesario que nuestras clases de error personalizadas hereden de error. Sin embargo, si heredan, será posible utilizar obj instanceof Error para identificar objetos de error. Por tanto, es mejor que hereden.

Además, cuando las aplicaciones crecen, nuestros errores pueden formar una jerarquía. Por ejemplo, que HttpTimeoutError herede de HttpError, que as su vez herede de Error.

## Extendiendo la clase Error

Por ejemplo, vamos a considerar una función readUser(json) que debería reer JSON con datos de usuario.

Un json válido sería similar a esto.

Internamente, utilizaremos JSON.parse. Si recibe un json mal formado, arrojará un SystaxError. Pero, auque el json sea sintácticamente correcto, esto no significa que sea un usuario correcto. Por ejemplo, si no tiene las propiedades name y age, que serían esenciales para los usuarios.

Nuestra función readUser(json) no leerá un JSON sino que validará los datos. Si el formato es correcto, pero no es un json de usuario, tendremos un ValidationError.

Nuestra clase ValidationError heredará de la clase Error.

La clase Error tiene un constructor que espera el mensaje del error, su código es similar a lo siguiente:

// Pseudocódigo para la clase Error

class Error {  
 constructor(message) {  
 this.message = message;  
 this.name = "Error"; // (name tendrá un valor diferente  
 // para cada tipo de Error  
 this.stack = <call stack>; // no-standard, pero muy soportado } }

Ahora, vamos a definir la clase ValidationError y a probarla:

class ValidationError extends Error {  
 constructor(message) {  
 super(message); // (1)  
 this.name = "ValidationError"; // (2)   
 }   
}

function test() {  
 throw new ValidationError("Whoops!");   
}

try {  
 test();   
} catch(err) {  
 alert(err.message); // Whoops!   
 alert(err.name); // ValidationError   
 alert(err.stack); // una lista de las llamadas anidadas con el número  
 // de línea de cada una  
}

Vamos a intentar utilizar readUser(json)

function readUser(json) {  
 let user = JSON.parse(json);

if (!user.age) {  
 throw new ValidationError("No campo: age");   
 }

if (!user.name) {  
 throw new ValidationError("No campo: name");   
 }  
 return user;   
}

// Working example with try..catch

try {   
 let user = readUser('{ "age": 25 }');   
} catch (err) {  
 if (err instanceof ValidationError) {// (\*)  
 alert("Dato inválido: " + err.message);//Dato inválido: No campo: name  
 } else if (err instanceof SyntaxError) {  
 alert("JSON Syntax Error: " + err.message);   
 } else {   
 throw err; // unknown error, rethrow it (\*\*)   
 }   
 }

Dentro del catch se gestiona tanto el error ValidationError como el error SyntaxERror, como cualquier otro error.

(\*) En lugar de (err instanceof ValidationError), podíamos haber utilizado (err.name==" ValidationError”).

Sin embargo, es preferible utilizar instanceOf, ya que la comparación devuelve true para esa clase y todas las subclases (si las hubiera).

(\*\*) Si catch encuentra un error desconocido (no es ValidationError, ni SystaxError), lo re-lanza.

## Más allá de la herencia

Supongamos que queremos crear una clase PropertyRequiredError, que indicaría que al json le falta alguna propiedad . Esta clase sería una subclase de ValidationError:

class ValidationError extends Error {  
 constructor(message) {  
 super(message);  
 this.name = "ValidationError";   
 }   
}  
  
class PropertyRequiredError extends ValidationError {  
 constructor(property) {  
 super("No property: " + property);  
 this.name = "PropertyRequiredError";  
 this.property = property;   
 }   
}  
  
 // Uso  
function readUser(json) {  
 let user = JSON.parse(json);   
 if (!user.age) {  
 throw new PropertyRequiredError("age");   
 }  
 if (!user.name) {  
 throw new PropertyRequiredError("name");   
 }   
 return user;   
}  
  
 // Working example with try..catch   
try {  
 let user = readUser('{ "age": 25 }');   
} catch (err) {  
 if (err instanceof ValidationError) {  
 alert("Dato inválido: " + err.message);//Dato inválido:No property:name   
 alert(err.name); // PropertyRequiredError   
 alert(err.property); // name   
 } else if (err instanceof SyntaxError) {  
 alert("JSON Syntax Error: " + err.message);   
 } else {  
 throw err; // unknown error, rethrow it   
 }   
}

En cada clase tenemos que definir la propiedad this.name.

Sería mejor generar una clase de error básica (que podemos llamar MyError) y asignar en ella this.name = this.constructor.name.

class MyError extends Error {   
 constructor(message) {  
 super(message);  
 this.name = this.constructor.name;   
 }   
}  
  
class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {  
 constructor(property) {  
 super("No property: " + property);  
 this.property = property;   
 }   
}