

Unidad 15

BackEnd - Node.JS

- 1 Uso y características de Express
- 2 Mostrar archivos por intermedio de sendfile
- 3 Mostrando archivos con node
- 4 Definiendo el motor de vistas con Engine.
- 5 .then y throw en el manejo de errores
- 6 haciendo llamada a funciones con diferentes parámetros
- 7 Uso de Callback en funciones Node
- 8 Instalación de Express
- 9 Api's Asíncronas
- 10 Archivos Estáticos
- 11 Manejo de Excepciones
- 12 Express y bases de datos.

Express, el framework mas utilizado de Node.Js

Express es el framework web más popular de Node, y es la librería subyacente para un gran número de otros frameworks web de Node populares. Proporciona mecanismos para:

- Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).
- Integración con motores de renderización de "vistas" para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
- Añadir procesamiento de peticiones "middleware" adicional en cualquier punto dentro de la tubería de manejo de la petición.

A pesar de que Express es en sí mismo bastante minimalista, los desarrolladores han creado paquetes de middleware compatibles para abordar casi cualquier problema de desarrollo web. Hay librerías para trabajar con cookies, sesiones, inicios de sesión de usuario, parámetros URL, datos POST, cabeceras de seguridad y muchos más. Podés encontrar una lista de paquetes middleware mantenida por el equipo de Express en Express Middleware (junto con una lista de algunos de los paquetes más populares de terceros).

¿Dónde comenzó?

Express fue lanzado inicialmente en Noviembre de 2010 y está ahora en la versión 4.17.1 de la API. Podés comprobar en el changelog la información sobre cambios en la versión actual, y en GitHub notas de lanzamiento históricas más detalladas.

¿Qué popularidad tiene Node/Express?

La popularidad de un framework web es importante porque es un indicador de si se continuará manteniendo y qué recursos tienen más probabilidad de estar disponibles en términos de documentación, librerías de extensiones y soporte técnico.

No existe una medida disponible de inmediato y definitiva de la popularidad de los frameworks de lado servidor (aunque sitios como Hot Frameworks intentan asesorar sobre popularidad usando mecanismos como contar para cada plataforma el número de preguntas sobre proyectos en GitHub y StackOverflow). Una pregunta mejor es si Node y Express son lo "suficientemente populares" para evitar los problemas de las plataformas menos populares. ¿Continúan evolucionando? ¿Podés conseguir la ayuda que necesitas? ¿Hay alguna posibilidad de que consigas un trabajo remunerado si aprendes Express?

De acuerdo con el número de compañías de perfil alto que usan Express, el número de gente que contribuye al código base, y el número de gente que proporciona soporte tanto libre como pagado, podemos entonces decir que Express es un framework extremadamente popular.

¿Es Express dogmático?

Los frameworks web frecuentemente se refieren a sí mismos como "dogmáticos" ("opinionated") o "no dogmáticos" ("unopinionated").

Los frameworks dogmáticos son aquellos que opinan acerca de la "manera correcta" de gestionar cualquier tarea en particular. Ofrecen soporte para el desarrollo rápido en un dominio en particular (resolver problemas de un tipo en particular) porque la manera correcta de hacer cualquier cosa está generalmente bien comprendida y bien documentada. Sin embargo pueden ser menos flexibles para resolver problemas fuera de su dominio principal, y tienden a ofrecer menos opciones para elegir qué componentes y enfoques pueden usarse.

Los frameworks no dogmáticos, en contraposición, tienen muchas menos restricciones sobre el modo mejor de unir componentes para alcanzar un objetivo, o incluso qué componentes deberían usarse. Hacen más fácil para los desarrolladores usar las herramientas más adecuadas para completar una tarea en particular, si bien al costo de que necesitas encontrar esos componentes por tí mismo.

Express es no dogmático, transigente. Podés insertar casi cualquier middleware compatible que te guste dentro de la cadena de manejo de la petición, en casi cualquier orden que te parezca. Podés estructurar la app en un fichero o múltiples ficheros y usar cualquier estructura de directorios. Es válido también mencionar que algunas veces podés sentir que tenes demasiadas opciones.

¿Cómo es el código para Express?

En sitios web o aplicaciones web dinámicas, que accedan a bases de datos, el servidor espera a recibir peticiones HTTP del navegador (o cliente). Cuando se recibe una petición, la aplicación determina cuál es la acción adecuada correspondiente, de acuerdo a la estructura de la URL y a la información (opcional) indicada en la petición con los métodos `POST` o `GET`. Dependiendo de la acción a realizar, puede que se necesite leer o escribir en la base de datos, o realizar otras acciones necesarias para atender la petición correctamente. La aplicación ha de responder al navegador, normalmente, creando una página HTML dinámicamente para él, en la que se muestre la información pedida, usualmente dentro de un elemento específico para este fin, en una plantilla HTML.

Express posee métodos para especificar que función ha de ser llamada dependiendo del verbo HTTP usado en la petición (`GET`, `POST`, `SET`, etc.) y la estructura de la URL ("ruta"). También tiene los métodos para especificar que plantilla ("view") o gestor de visualización utilizar, donde están guardadas las plantillas de HTML que han de usarse y como generar la visualización adecuada para cada caso. El middleware de Express, puede usarse también para añadir

Copy on Clipboard para la gestión de cookies, sesiones y usuarios, mediante el uso de parámetros, en los métodos `POST/GET`.

Puede utilizarse además cualquier sistema de trabajo con bases de datos, que sea soportado por Node (Express no especifica ningún método preferido para trabajar con bases de datos).

En las siguientes secciones, se explican algunos puntos comunes que se pueden encontrar cuando se trabaja con código de Node y Express.

Hola Mundo! - en Express

Primero consideremos el tradicional ejemplo de Hola Mundo! (se comentará cada parte a continuación).

```
var express = require('express');

var app = express();

app.get('/', function(req, res) {

  res.send('Hola Mundo!');

});

app.listen(3000, function() {

  console.log('Aplicación ejemplo, escuchando el puerto 3000!');

});
```

Las primeras dos líneas incluyen (mediante la orden `require()`) el módulo de Express y crean una aplicación de Express. Este elemento se denomina comúnmente `app`, y posee métodos para el enrutamiento de las peticiones HTTP, configuración del 'middleware', y visualización de las vistas de HTML, uso del motores de 'templates', y gestión de las configuraciones de las aplicaciones que controlan la aplicación (por ejemplo el entorno, las definiciones para enrutado ... etcetera.)

Las líneas que siguen en el código (las tres líneas que comienzan con `app.get`) muestran una definición de ruta que se llamará cuando se reciba una petición HTTP `GET` con una dirección `('/')` relativa al directorio raíz. La función 'callback' toma una petición y una respuesta como argumentos, y ejecuta un `send()` en la respuesta, para enviar la cadena de caracteres: "Hola Mundo!".

El bloque final de código, define y crea el servidor, escuchando el puerto 3000 e imprime un comentario en la consola. Cuando se está ejecutando el servidor, es posible ir hasta la dirección `localhost:3000` en un navegador, y ver como el servidor de este ejemplo devuelve el mensaje de respuesta.

Instalando el módulo Express

Instalación

Suponiendo que ya ha instalado Node.js, creá un directorio para que contenga la aplicación y convertilo en el directorio de trabajo.

```
$ mkdir myapp  
$ cd myapp
```

Utilizá el comando `npm init` para crear un archivo `package.json` para la aplicación.

```
$ npm init
```

Este comando solicita varios elementos como, por ejemplo, el nombre y la versión de la aplicación. Por ahora, sólo tenés que pulsar **INTRO** para aceptar los valores predeterminados para la mayoría de ellos, con la siguiente excepción:

```
entry point: (index.js)
```

Especificá `app.js` o el nombre que desees para el archivo principal. Si desees que sea `index.js`, pulsás enter y listo, para aceptar el nombre de archivo predeterminado recomendado.

A continuación, instalá Express en el directorio `myapp` y guardalo en la lista de dependencias. Por ejemplo:

```
$ npm install express --save
```

Para instalar Express temporalmente y no añadirlo a la lista de dependencias, omití la opción `--save`:

```
$ npm install express
```

Los módulos de Node que se instalan con la opción `--save` se añaden a la lista `dependencies` en el archivo `package.json`. Posteriormente, si ejecutás `npm install` en el directorio `app`, los módulos se instalarán automáticamente en la lista de dependencias.

Función Callback

Una función de callback es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción.

Ejemplo:

```
function saludar(nombre) {  
  
    alert('Hola ' + nombre);  
  
}  
  
function procesarEntradaUsuario(callback) {  
  
    var nombre = prompt('Por favor ingresa tu nombre.');
```



```
    callback(nombre);  
  
}  
  
procesarEntradaUsuario(saludar);
```

El ejemplo anterior es una callback sincrónica, ya que se ejecuta inmediatamente.

Sin embargo, tené en cuenta que las callbacks a menudo se utilizan para continuar con la ejecución del código después de que se haya completado una operación a sincrónica — estas se denominan devoluciones de llamada asincrónicas.

Importando y creando módulos

Un modulo es una librería o archivo JavaScript que puede ser importado dentro de otro código utilizando la función `require()` de Node. Por sí mismo, Express es un modulo, como lo son el middleware y las librerías de bases de datos que se utilizan en las aplicaciones Express.

El código mostrado abajo, muestra como puede importarse un modulo con base a su nombre, como ejemplo se utiliza el framework Express. Primero se invoca la función `require()`, indicando como parámetro el nombre del módulo o librería como una cadena ('express'), posteriormente se invoca el objeto obtenido para crear una [aplicación Express](#).

Posteriormente, se puede acceder a las propiedades y funciones del objeto Aplicación.

```
var express = require('express');  
  
var app = express();
```

También podemos crear nuestros propios módulos que puedan posteriormente ser importados de la misma manera.

Para hacer que los objetos esten disponibles fuera de un modulo, solamente es necesario asignarlos al objeto `exports`. Por ejemplo, el modulo mostrado a continuación `square.js` es un archivo que exporta los métodos `area()` y `perimeter()` :

```
exports.area = function(width) { return width * width; };  
  
exports.perimeter = function(width) { return 4 * width; };
```

Nosotros podemos importar este módulo utilizando la función `require()`, y entonces podremos invocar los métodos exportados de la siguiente manera:

```
// Utilizamos la función require() El nombre del archivo se ingresa sin la extensión (opcional) .js  
  
var square = require('./square');  
  
// invocamos el metodo area()  
  
console.log('El área de un cuadrado con lado de 4 es ' + square.area(4));
```


Si querés exportar completamente un objeto en una asignación en lugar de construir cada propiedad por separado, debes asignarlo al módulo `module.exports` como se muestra a continuación (también podés hacer esto al inicio de un constructor o de otra función.)

```
module.exports = {  
  
  area: function(width) {  
  
    return width * width;  
  
  },  
  
  perimeter: function(width) {  
  
    return 4 * width;  
  
  }  
  
};
```

Usando APIs asíncronas

El código JavaScript usa frecuentemente APIs asíncronas antes que síncronas para operaciones que tomen algún tiempo en completarse. En una API síncrona cada operación debe completarse antes de que la siguiente pueda comenzar. Por ejemplo, la siguiente función de registro es síncrona, y escribirá en orden el texto en la consola (Primero, Segundo).

```
console.log('Primero');  
  
console.log('Segundo');
```

En contraste, en una API asíncrona, la API comenzará una operación e inmediatamente retornará (antes de que la operación se complete). Una vez que la operación finalice, la API usará algún mecanismo para realizar operaciones adicionales. Por ejemplo, el código de abajo imprimirá "Segundo, Primero" porque aunque el método `setTimeout()` es llamado primero y retorna inmediatamente, la operación no se completa por varios segundos.

```
setTimeout(function() {  
  
    console.log('Primero');  
  
}, 3000);  
  
console.log('Segundo');
```

Usar APIs asíncronas sin bloques es aun mas importante en Node que en el navegador, porque Node es un entorno de ejecución controlado por eventos de un solo hilo. "Un solo hilo" quiere decir que todas las peticiones al servidor son ejecutadas en el mismo hilo (en vez de dividirse en procesos separados). Este modelo es extremadamente eficiente en términos de velocidad y recursos del servidor, pero eso significa que si alguna de sus funciones llama a métodos síncronos que tomen demasiado tiempo en completarse, bloquearan no solo la solicitud actual, sino también cualquier otra petición que este siendo manejada por tu aplicación web.

Hay muchas maneras para una API asíncrona de notificar a su aplicación que se ha completado. La manera mas común es registrar una función callback cuando invocás a una API asíncrona, la misma será llamada de vuelta cuando la operación se complete. Éste es el enfoque utilizado anteriormente.

En el siguiente ejemplo de "Hola Mundo!" en Express, definimos una función (callback) manejadora de ruta para peticiones HTTP `GET` a la raíz del sitio (`/`).

```
app.get('/', function(req, res) {  
  
  res.send('Hello World!');  
  
});
```

La función callback toma una petición y una respuesta como argumentos. En este caso el método simplemente llama a `send()` en la respuesta para retornar la cadena "Hello World!". Hay un número de otros métodos de respuesta para finalizar el ciclo de solicitud/respuesta, por ejemplo podrá llamar a `res.json()` para enviar una respuesta JSON o `res.sendFile()` para enviar un archivo.

El objeto que representa una aplicación de Express, también posee métodos para definir los manejadores de rutas para el resto de los verbos HTTP: `post()`, `put()`, `delete()`, `options()`, `trace()`, `copy()`, `lock()`, `mkcol()`, `move()`, `purge()`, `propfind()`, `proppatch()`, `unlock()`, `report()`, `mkactivity()`, `checkout()`, `merge()`, `m-search()`, `notify()`, `subscribe()`, `unsubscribe()`, `patch()`, `search()`, y `connect()`.

Hay un método general para definir las rutas: `app.all()`, el cual será llamado en respuesta a cualquier método HTTP. Se usa para cargar funciones del middleware en una dirección particular para todos los métodos de peticiones. El siguiente ejemplo (de la documentación de Express) muestra el uso de los manejadores a `/secret` sin tener en cuenta el verbo HTTP utilizado (siempre que esté definido por el módulo `http`).

```
app.all('/secret', function(req, res, next) {  
  
  console.log('Accediendo a la seccion secreta ...');  
  
  next(); // pasa el control al siguiente manejador  
  
});
```

Las rutas le permiten igualar patrones particulares de caracteres en la URL, y extraer algunos valores de ella y pasarlos como parámetros al manejador de rutas (como atributo del objeto petición pasado como parámetro).

Usualmente es útil agrupar manejadores de rutas para una parte del sitio juntos y accederlos usando un prefijo de ruta en común. (Ej: un sitio con una Wiki podría tener todas las rutas relacionadas a dicha sección en un archivo y siendo accedidas con el prefijo de ruta `/wiki/`. En Express esto se logra usando el objeto `express.Router`. Ej: podemos crear nuestra ruta wiki en un módulo llamado `wiki.js`, y entonces exportar el objeto `Router`, como se muestra debajo:

```
// wiki.js - Modulo de rutas Wiki
```

```
var express = require('express');

var router = express.Router();

// Home

router.get('/', function(req, res) {

    res.send('Página de inicio Wiki');

});

// About page route

router.get('/about', function(req, res) {

    res.send('Acerca de esta wiki');

});

module.exports = router;
```

Para usar el router en nuestro archivo app principal, necesitamos `require()` el módulo de rutas (`wiki.js`), entonces llamar a `use()` en la aplicación Express para agregar el Router al software intermediario que maneja las rutas. Las dos rutas serán accesibles entonces desde `/wiki/` y `/wiki/about/`.

```
var wiki = require('./wiki.js');

// ...

app.use('/wiki', wiki);
```

Usando middleware

El "middleware" es ampliamente utilizado en las aplicaciones de Express: desde tareas para servir archivos estáticos, a la gestión de errores o la compresión de las respuestas HTTP. Mientras las

funciones de enrutamiento, con el objeto `express.Router`, se encargan del ciclo petición-respuesta, al gestionar la respuesta adecuada al cliente, las funciones de middleware normalmente realizan alguna operación al gestionar una petición o respuesta y a continuación llaman a la siguiente función en la "pila", que Podés ser otra función de middleware u otra función de enrutamiento. El orden en el que las funciones de middleware son llamadas depende del desarrollador de la aplicación.

La mayoría de las aplicaciones usan middleware desarrollado por terceras partes, para simplificar funciones habituales en el desarrollo web, como Podés ser: gestión de cookies, sesiones, autenticado de usuarios, peticiones `POST` y datos en JSON, registros de eventos, etc. Podés encontrar en `npm.js` una lista de middleware mantenido por el equipo de Express (que también incluye otros paquetes populares de terceras partes). Las librerías de Express están disponibles con la aplicación NPM (Node Package Manager).

Para usar estas colecciones, primero ha de instalar la aplicación usando NPM. Por ejemplo para instalar el registro de peticiones HTTP morgan, se haría con el comando Bash:

```
$ npm install morgan
```

Entonces podrías llamar a la función `use()` en un objeto de aplicación Express para utilizar este middleware a su aplicación.

```
var express = require('express');

var logger = require('morgan');

var app = express();

app.use(logger('dev'));

...
```

Podés escribir tu propia función middleware, y si quieres hacerlo así (solo para crear código de manejo de error). La única diferencia entre una función middleware y un callback manejador de rutas es que las funciones middleware tienen un tercer argumento `next`, cuyas funciones middleware son esperadas para llamarlas si ellas no completan el ciclo request (cuando la función middleware es llamada, esta contiene la próxima función que debe ser llamada).

Podés agregar una función middleware a la cadena de procesamiento con cualquier `app.use()` o `app.add()`, dependiendo de si hay que aplicar el middleware a todas las

respuestas o a respuestas con un verbo particular HTTP (GET, POST, etc). Vos especificas las rutas, lo mismo en ambos casos, aunque la ruta es opcional cuando llama `app.use()`.

El ejemplo de abajo muestra como podés agregar la función middleware usando ambos métodos, y con/sin una ruta.

```
var express = require('express');

var app = express();

// An example middleware function

var a_middleware_function = function(req, res, next) {

  // ... perform some operations

  next(); // Call next() so Express will call the next middleware function in the chain.
}

// Function added with use() for all routes and verbs


app.use(a_middleware_function);

// Function added with use() for a specific route

app.use('/someroute', a_middleware_function);

// A middleware function added for a specific HTTP verb and route

app.get('/', a_middleware_function);
```



```
app.listen(3000);
```

Sirviendo archivos estáticos

Podés utilizar el middleware `express.static` para servir archivos estáticos, incluyendo sus imágenes, CSS y JavaScript (`static()` es la única función middleware que es actualmente parte de Express). Por ejemplo, podrias utilizar la linea de abajo para servir imágenes, archivos CSS, y archivos JavaScript desde un directorio nombrado 'public' al mismo nivel desde donde llama a node:

```
app.use(express.static('public'));
```

Cualquier archivo existente en el directorio público son servidos al agregar su nombre de archivo (relativo a la ubicación del directorio "público") de la ubicación URL. Por ejemplo:

```
http://localhost:3000/images/dog.jpg
```

```
http://localhost:3000/css/style.css
```

```
http://localhost:3000/js/app.js
```

```
http://localhost:3000/about.html
```

Podés llamar `static()` en multiples ocasiones a servir multiples directorios. Si un archivo no puede ser encontrado por una función middleware entonces éste simplemente será pasado en la subsecuente middleware (el orden en que el middleware está basado en su orden de declaración).

```
app.use(express.static('public'));
```

```
app.use(express.static('media'));
```

Tambien podés crear un prefijo virtual para sus URLs estáticas, aun más teniendo los archivos agregados en la ubicación URL. Por ejemplo, aqui especificamos a mount path tal que los archivos son bajados con el prefijo `"/media"`:

```
app.use('/media', express.static('public'));
```

Ahora, Podés bajar los archivos que estan en el directorio publico del path con prefijo `/media`.

```
http://localhost:3000/media/images/dog.jpg
```

```
http://localhost:3000/media/video/cat.mp4
```

```
http://localhost:3000/media/cry.mp3
```


Manejando errores

Los errores son manejados por una o más funciones especiales middleware que tienen cuatro argumentos, en lugar de las usuales tres: (err, req, res, next). For example:

```
app.use(function(err, req, res, next) {  
  
  console.error(err.stack);  
  
  res.status(500).send('Something broke!');  
  
});
```

Estas pueden devolver cualquier contenido, pero deben ser llamadas después de todas las otras `app.use()` llamadas de rutas tal que ellas son las últimas middleware en el proceso de manejo de request.

Express viene con un manejador de error integrado, el que se ocupa de cualquier error remanente que pudiera ser encontrado en la app. Esta función middleware con su manejador de error esta agregada al final del stack de funciones middleware. Si pasa un error a `next()` y no lo maneja en un manejador de error, éste sera manejado por el manejador de error integrado; el error sera escrito en el cliente con el rastreo de pila.

Expresiones de manejo de excepciones

Podés lanzar excepciones usando la instrucción `throw` y manejarlas usando las declaraciones `try...catch`.

- Expresión `throw`
- Declaración `try...catch`

Tipos de excepciones

Casi cualquier objeto se puede lanzar en JavaScript. Sin embargo, no todos los objetos lanzados son iguales. Si bien es común lanzar números o cadenas como errores, con frecuencia es más efectivo usar uno de los tipos de excepción creados específicamente para este propósito:

- excepciones ECMAScript
- La interfaz `DOMException` (en-US) representa un evento anormal (llamado excepción) que ocurre como resultado de llamar a un método o acceder a una propiedad de una API web y la interfaz `DOMError` describe un objeto de error que contiene un nombre de error.

Expresión `throw`

Utilizá la expresión `throw` para lanzar una excepción. Una expresión `throw` especifica el valor que se lanzará:

```
throw expression;
```

Podés lanzar cualquier expresión, no solo expresiones de un tipo específico. El siguiente código arroja varias excepciones de distintos tipos:

```
throw 'Error2'; // tipo String

throw 42;       // tipo Number

throw true;     // tipo Boolean

throw {toString: function() { return "¡Soy un objeto!"; } };
```

Nota Podés especificar un objeto cuando lanzas una excepción. A continuación, Podés hacer referencia a las propiedades del objeto en el bloque `catch`.

```
// Crea un objeto tipo de UserException

function UserException(message) {
```

```
this.message = message;

this.name = 'UserException';

}

// Hacer que la excepción se convierta en una cadena

// (por ejemplo, por la consola de errores)

UserException.prototype.toString = function() {

    return `${this.name}: "${this.message}"`;

}

// Crea una instancia del tipo de objeto y tirla

throw new UserException('Valor muy alto');
```

Declaración try...catch

La declaración try...catch marca un bloque de expresiones para probar y especifica una o más respuestas en caso de que se produzca una excepción. Si se lanza una excepción, la declaración try...catch la detecta.

La declaración try...catch consta de un bloque try, que contiene una o más declaraciones, y un bloque catch, que contiene declaraciones que especifican qué hacer si se lanza una excepción en el bloque try.

En otras palabras, si deseas que el bloque try tenga éxito, pero si no es así, deseas que el control pase al bloque catch. Si alguna instrucción dentro del bloque try (o en una función llamada desde dentro del bloque try) arroja una excepción, el control inmediatamente cambia al bloque catch. Si no se lanza ninguna excepción en el bloque try, se omite el bloque catch. El bloque finalmente se ejecuta después de que se ejecutan los bloques try y catch, pero antes de las declaraciones que siguen a la declaración try...catch.

El siguiente ejemplo usa una instrucción `try...catch`. El ejemplo llama a una función que recupera el nombre de un mes de un arreglo en función del valor pasado a la función. Si el valor no corresponde a un número de mes (1-12), se lanza una excepción con el valor "Mes Inválido" y las declaraciones en el bloque `catch` establezca la variable `monthName` en 'unknown'.

```
function getMonthName(mo) {

    mo = mo - 1; // Ajusta el número de mes para el índice del arreglo (1 = Ene, 12 = Dic)

    let months = ['Ene', 'Feb', 'Mar', 'Abr', 'May', 'Jun', 'Jul',

                  'Ago', 'Sep', 'Oct', 'Nov', 'Dic'];

    if (months[mo]) {

        return months[mo];

    } else {

        throw 'Mes Inválido'; // aquí se usa la palabra clave throw

    }

}

try { // declaraciones para try

    monthName = getMonthName(myMonth); // la función podría lanzar una excepción

}

catch (e) {

    monthName = 'unknown';

    logMyErrors(e); // pasar el objeto exception al controlador de errores (es decir, su propia función)
```

```
}
```

El bloque `catch`

Podés usar un bloque `catch` para manejar todas las excepciones que se puedan generar en el bloque `try`.

```
catch (catchID) {  
  
    instrucciones  
  
}
```

El bloque `catch` especifica un identificador (`catchID` en la sintaxis anterior) que contiene el valor especificado por la expresión `throw`. Podés usar este identificador para obtener información sobre la excepción que se lanzó.

JavaScript crea este identificador cuando se ingresa al bloque `catch`. El identificador dura solo la duración del bloque `catch`. Una vez que el bloque `catch` termina de ejecutarse, el identificador ya no existe.

Por ejemplo, el siguiente código lanza una excepción. Cuando ocurre la excepción, el control se transfiere al bloque `catch`.

```
try {  
  
    throw 'myException'; // genera una excepción  
  
}  
  
catch (err) {  
  
    // declaraciones para manejar cualquier excepción  
  
    logMyErrors(err); // pasa el objeto exception al controlador de errores  
  
}
```

El bloque `finally`

El bloque `finally` contiene instrucciones que se ejecutarán después que se ejecuten los bloques `try` y `catch`. Además, el bloque `finally` ejecuta antes el código que sigue a la declaración `try...catch...finally`.

También es importante notar que el bloque `finally` se ejecutará independientemente de que se produzca una excepción. Sin embargo, si se lanza una excepción, las declaraciones en el bloque `finally` se ejecutan incluso si ningún bloque `catch` maneje la excepción que se lanzó.

Podés usar el bloque `finally` para hacer que tu script falle correctamente cuando ocurra una excepción. Por ejemplo, es posible que debas liberar un recurso que tu script haya inmovilizado.

El siguiente ejemplo abre un archivo y luego ejecuta declaraciones que usan el archivo. (JavaScript de lado del servidor te permite acceder a los archivos). Si se lanza una excepción mientras el archivo está abierto, el bloque `finally` cierra el archivo antes de que falle el script.

Usar `finally` aquí asegura que el archivo nunca se deje abierto, incluso si ocurre un error.

```
openMyFile();

try {

    writeMyFile(theData); // Esto puede arrojar un error

} catch(e) {

    handleError(e); // Si ocurrió un error, manéjalo

} finally {

    closeMyFile(); // Siempre cierra el recurso

}
```

Si el bloque `finally` devuelve un valor, este valor se convierte en el valor de retorno de toda la producción de `try...catch...finally`, independientemente de las declaraciones `return` en los bloques `try` y `catch`:

```
function f() {

    try {
```

```
console.log(0);

throw 'bogus';

} catch(e) {

    console.log(1);

    return true; // esta declaración de retorno está suspendida

                // hasta que el bloque finally se haya completado

    console.log(2); // no alcanzable

} finally {

    console.log(3);

    return false; // sobrescribe el "return" anterior

    console.log(4); // no alcanzable

}

// "return false" se ejecuta ahora

console.log(5); // inalcanzable

}

console.log(f()); // 0, 1, 3, false
```

La sobrescritura de los valores devueltos por el bloque `finally` también se aplica a las excepciones lanzadas o relanzadas dentro del bloque `catch`:

```
function f() {

    try {
```

```
    throw 'bogus';

} catch(e) {

    console.log('captura "falso" interno');

    throw e; // esta instrucción throw se suspende hasta

               // que el bloque finally se haya completado

} finally {

    return false; // sobrescribe el "throw" anterior

}

// "return false" se ejecuta ahora

}

try {

    console.log(f());

} catch(e) {

    // ¡esto nunca se alcanza!

    // mientras se ejecuta f(), el bloque `finally` devuelve false,

    // que sobrescribe el `throw` dentro del `catch` anterior

    console.log('"falso" externo capturado');

}
```



```
// Produce
```

```
// "falso" interno capturado
```

```
// false
```

Declaraciones `try...catch` anidadas

Podés anidar una o más declaraciones `try...catch`.

Si un bloque `try` interno no tiene un bloque `catch` correspondiente:

1. debe contener un bloque `finally`, y
2. el bloque `catch` adjunto de la declaración `try...catch` se comprueba para una coincidencia.

Para obtener más información, consulta bloques `try` anidados en la una página de referencia `try...catch`.

Utilizar objetos `Error`

Dependiendo del tipo de error, es posible que puedas utilizar las propiedades `name` y `message` para obtener un mensaje más refinado.

La propiedad `name` proporciona la clase general de `Error` (tal como `DOMException` o `Error`), mientras que `message` generalmente proporciona un mensaje más conciso que el que se obtendría al convertir el objeto error en una cadena.

Si estás lanzando tus propias excepciones, para aprovechar estas propiedades (por ejemplo, si tu bloque `catch` no discrimina entre tus propias excepciones y las del sistema), Podés usar el constructor `Error`.

Por ejemplo:

```
function doSomethingErrorProne() {  
  
    if (ourCodeMakesAMistake()) {  
  
        throw (new Error('El mensaje'));  
  
    } else {  
  
        doSomethingToGetAJavascriptError();  
  
    }  
}
```

```
}  
  
:  
  
try {  
  
    doSomethingErrorProne();  
  
} catch (e) {          // AHORA, en realidad usamos `console.error()`  
  
    console.error(e.name); // registra 'Error'  
  
    console.error(e.message); // registra 'The message' o un mensaje de error de JavaScript  
  
}
```

Usando Bases de datos

Las apps de Express pueden usar cualquier mecanismo de bases de datos soportadas por Node (Express en sí mismo no define ninguna conducta/requerimiento específico adicional para administración de bases de datos). Hay muchas opciones, incluyendo PostgreSQL, MySQL, Redis, SQLite, MongoDB, etc.

Con el propósito de usar éste, primero hay instalar el manejador de bases de datos utilizando NPM. Por ejemplo, para instalar el manejador para el popular NoSQL MongoDB querría utilizar el comando:

```
$ npm install mongodb
```

La base de datos por si misma puede ser instalada localmente o en un servidor de la nube. En su código Express requiere el manejador, conectarse a la base de datos, y entonces ejecutar operaciones crear, leer, actualizar, y borrar (CLAB). El ejemplo de abajo (de la documentación Express documentation) muestra como puede encontrar registros en la colección "mamiferos" usando MongoDB.

```
var MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017/animals', function(err, db) {

  if (err) throw err;

  db.collection('mammals').find().toArray(function (err, result) {

    if (err) throw err;

    console.log(result);

  });

});
```

Otra aproximación es acceder a su base de datos indirectamente, via algún Mapeo Objeto Relacional ("MOR"). En esta aproximación se define sus datos como "objetos" o "modelos" y el MOR mapea estos a través del delineamiento básico de la base de datos. Esta aproximación tiene el beneficio de que como un desrollador puede continuar pensando en términos de objetos de JavaScript mas que en semántica de bases de datos, y en esto hay un lugar obvio para ejecutar la validación y chequeo de entrada de datos. Hablaremos más de bases de datos en un artículo posterior.

Renderización de data (vistas)

El Motor de plantilla (referido como "motor de vistas" por Express) le permite definir la estructura de documento de salida en una plantilla, usando marcadores de posición para datos que serán llenados cuando una página es generada. Las plantillas son utilizadas generalmente para crear HTML, pero también pueden crear otros tipos de documentos.

En su código de configuración de su aplicación vos configuras el motor de plantillas para usar y su localización Express podría buscar plantillas usando las configuraciones de 'vistas' y 'motores de vistas', mostrado abajo (tendría también que instalar el paquete conteniendo su librería de plantillas)

```
var express = require('express');

var app = express();

// Set directory to contain the templates ('views')

app.set('views', path.join(__dirname, 'views'));

// Set view engine to use, in this case 'some_template_engine_name'

app.set('view engine', 'some_template_engine_name');
```

La apariencia de la plantilla dependerá de qué motor use. Asumiendo que tiene un archivo de plantillas nombrado "index.<template_extension>" este contiene placeholders para variables de datos nombradas 'title' y "message", podría llamar `Response.render()` en una función manejadora de rutas para crear y enviar la HTML response:

```
app.get('/', function(req, res) {
```

```
res.render('index', { title: 'About dogs', message: 'Dogs rock!' });  
  
});
```

Estructura de Archivos

Express no hace las cargas de estructura o de componentes utilizados. Rutas, vistas, archivos estáticos, y otras lógicas de aplicación específica pueden vivir en cualquier número de archivos con cualquier estructura de directorio. Mientras que esto es perfectamente posible, se puede tener toda la aplicación en un solo archivo. En Express, típicamente esto tiene sentido al desplegar tu aplicación dentro de archivos basados en función (e.g. administracion de cuentas, blogs, tableros de discusion) y dominio de problema arquitectonico (e.g. modelo, vista or controlador si pasás a estar usando una arquitectura MVC).