Node.js

https://nodejs.org/es/
https://nodejs.dev/

INTRODUCCIÓN

Introducción

- Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.
- Al contrario que la mayoría del código JavaScript, no se ejecuta en un navegador, sino en el servidor.
- Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web.
- Fue creado por Ryan Dahl en 2009 y su evolución está apadrinada por la empresa Joyent, que además tiene contratado a Dahl en plantilla.
- Node.js tiene un diseño similar y está influenciado por sistemas como Event Machine de Ruby ó Twisted de Python.
- Node.js implementa algunas especificaciones de CommonJS.
- Node.js incluye un entorno REPL para depuración interactiva.

Orientado a eventos asíncronos

- Concebido como un entorno de ejecución de JavaScript orientado a eventos asíncronos, Node.js está diseñado para construir aplicaciones en red escalables, que puedan manejar muchas conexiones concurrentes. Por cada conexión, el callback será ejecutado, sin embargo, si no hay trabajo que hacer, Node.js estará durmiendo.
- Esto contrasta con el modelo de concurrencia más común hoy en día, donde se usan hilos del Sistema Operativo.
- Las operaciones de redes basadas en hilos son relativamente ineficientes y son muy difíciles de usar.
- Además, los desarrolladores de Node.js están libres de preocupaciones sobre el bloqueo del proceso, ya que no existe. Casi ninguna función en Node.js realiza I/O directamente, así que el proceso nunca se bloquea. Debido a que no hay bloqueo es muy razonable desarrollar sistemas escalables en Node.js.

Modelo de eventos

- Node.js lleva el modelo de eventos un poco más allá, este presenta un bucle de eventos como un entorno en vez de una librería.
- En otros sistemas siempre existe una llamada que bloquea para iniciar el bucle de eventos. El comportamiento es típicamente definido a través de callbacks al inicio del script y al final se inicia el servidor mediante una llamada de bloqueo como EventMachine::run().
- En Node.js no existe esta llamada: Node.js simplemente ingresa en el bucle de eventos después de ejecutar el script de entrada.
- Node.js sale del bucle de eventos cuando no hay más callbacks que ejecutar.
- Se comporta de una forma similar a JavaScript en el navegador: el bucle de eventos está oculto al usuario.

Concurrencia y eventos

- Node.js funciona con un modelo de evaluación de un único hilo de ejecución, usando entradas y salidas asíncronas las cuales pueden ejecutarse concurrentemente en un número de hasta cientos de miles sin incurrir en costos asociados al cambio de contexto.
- Este diseño de compartir un único hilo de ejecución entre todas las solicitudes atiende a necesidades de aplicaciones altamente concurrentes, en el que toda operación que realice entradas y salidas debe tener una función callback.
- Un inconveniente de este enfoque de único hilo de ejecución es que Node.js requiere de módulos adicionales como cluster para escalar la aplicación con el número de núcleos de procesamiento de la máquina en la que se ejecuta.
- Node.js se registra con el sistema operativo y cada vez que un cliente establece una conexión se ejecuta un callback. Dentro del entorno de ejecución de Node.js, cada conexión recibe una pequeña asignación de espacio de memoria dinámico, sin tener que crear un hilo de ejecución.
- A diferencia de otros servidores dirigidos por eventos, el bucle de gestión de eventos de Node.js no es llamado explícitamente, sino que se activa al final de cada ejecución de una función callback. El bucle de gestión de eventos se termina cuando ya no quedan eventos por atender.

V8

- V8 es el motor de código abierto de alto rendimiento de JavaScript y WebAssembly de Google, escrito en C ++. Se usa en Chrome y en Node.js, entre otros.
- Implementa ECMAScript y WebAssembly, y se ejecuta en Windows 7 o posterior, macOS 10.12+ y sistemas Linux que usan procesadores x64, IA-32, ARM o MIPS.
- V8 puede ejecutarse de forma independiente o puede integrarse en cualquier aplicación C ++.
- Es software libre desde 2008 y compila el código fuente JavaScript en código de máquina en lugar de interpretarlo en tiempo real.
- Node.js contiene libuv para manejar eventos asíncronos. Libuv es una capa de abstracción de funcionalidades de redes y sistemas de archivo en sistemas Windows y sistemas basados en POSIX como Linux, Mac OS X y Unix.
- El cuerpo de operaciones de base de Node.js está escrito en JavaScript con métodos de soporte escritos en C++.

Módulos

- Node.js incorpora varios "módulos básicos" compilados en el propio binario, como por ejemplo el módulo de red, que proporciona una capa para programación de red asíncrona y otros módulos fundamentales, como por ejemplo HTTP, Path, FileSystem, Buffer, Timers y el de propósito más general Stream.
- Es posible utilizar módulos desarrollados por terceros, ya sea como archivos ".node" pre-compilados o como archivos en JavaScript. Los módulos JavaScript se implementan siguiendo la especificación CommonJS para módulos, utilizando una variable de exportación para dar a estos scripts acceso a funciones y variables implementadas por los módulos.
- Los módulos de terceros pueden extender node.js o añadir un nivel de abstracción, implementando varias utilidades middleware para utilizar en aplicaciones web, como por ejemplo los frameworks connect y express.
- Pese a que los módulos pueden instalarse como archivos simples, normalmente se instalan utilizando el Node Package Manager (npm) que nos facilitará la compilación, instalación y actualización de módulos así como la gestión de las dependencias.

Full Stack

- Node.js permite el desarrollo del back-end en JavaScript.
- Node.js puede ser combinado con una base de datos documental (por ejemplo, MongoDB o CouchDB) y JSON.
- Esto permite desarrollar en un entorno de desarrollo JavaScript unificado, un desarrollo homogéneo entre cliente, servidor y base de datos si cambios de contexto. Aunque la creación de aplicaciones que se ejecutan en el navegador es algo completamente diferente a la creación de una aplicación Node.js.
- Con la adaptación de los patrones para desarrollo del lado del servidor tales como MVC y sus variantes MVP, MVVM, etc. Node.js facilita la reutilización de código del mismo modelo de interfaz entre el lado del cliente y el lado del servidor.

HERRAMIENTAS DE DESARROLLO

IDEs

- Visual Studio Code
  - http://code.visualstudio.com/
- Gitpod
  - https://www.gitpod.io/
- StackBlitz
  - https://stackblitz.com
- Visual Studio 2017+
  - https://visualstudio.microsoft.com/es/downloads/
- Eclipse with Eclipse Wild Web Developer extension
  - https://www.eclipse.org/eclipseide/
- Webstorm
  - https://www.jetbrains.com/webstorm/

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: https://git-scm.com/
- Verificar desde consola de comandos:
  - git

Node.js: Entorno en tiempo de ejecución

- Las nuevas releases major de Node.js se sacan de la rama master de GitHub cada seis meses. Las versiones pares se sacan en abril, y las impares en octubre. Cuando se libera una versión impar, la versión par anterior pasa a soporte a largo plazo (Long Term Support, LTS), que da a la versión un soporte activo de 18 meses desde la fecha de inicio de la LTS. Después de estos 18 meses, la versión recibe otros 12 meses de soporte de mantenimiento. Una versión activa recibirá los cambios compatibles unas pocas semanas después de que aterricen en la versión estable actual. Una versión de mantenimiento recibirá sólo actualizaciones críticas y de documentación.
- Descargar e instalar: https://nodejs.org
- Verificar desde consola de comandos:
  - node --version

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
  - npm update -g npm
- Verificar desde consola de comandos:
  - npm --version
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

npx

- npx es un ejecutor de paquetes binarios de npm.
- npx es una herramienta destinada a ayudar a completar la experiencia de usar paquetes del registro de npm: de la misma manera que npm hace que sea muy fácil instalar y administrar dependencias alojadas en el registro, npx facilita el uso de herramientas CLI y otros ejecutables alojados en el registro.
- Para evitar tener que instalar globalmente herramientas de uso infrecuente pero que cambian continuamente, npx descarga, ejecuta y descarta las herramientas sin requerir instalación.
- Disponible desde la versión npm@5.2.0, se puede instalar manualmente con:
  - npm i –g npx
- Para ejecutar un comando al vuelo:
  - npx create-react-app my-react-app

nvm-windows

- Hay situaciones en las que la capacidad de cambiar entre diferentes versiones de Node.js puede ser muy útil. Por ejemplo, si se desea probar un módulo que se está desarrollando con la última versión de última generación sin desinstalar la versión estable del Node.
- Node Version Manager (nvm) for Windows permite administrar múltiples instalaciones de Node.js en una maquina con Windows.
- Descargar e instalar (es conveniente no tener instalado previamente Node): https://github.com/coreybutler/nvm-windows/releases
- Para conocer las versiones de Node disponibles para instalar:
  - nvm list available
- Para instalar una versión de Node:
  - nvm install <version>
- Para saber que versiones de Node hay instaladas:
  - nvm list
- Para usar una versión de Node:
  - nvm use <version>
- Para desinstalar una versión de Node:
  - nvm uninstall <version>

nodemon

- En tiempo de desarrollo, cada vez que se guardan los cambios de una aplicación de Node.js es necesario parar el servidor y volver a arrancarlo.
- Estar constantemente reiniciando manualmente es un trabajo muy tedioso y también agotador, pero para evitar tener que realizar este trabajo una y otra vez, existe Nodemon que se encarga de vigilar el directorio de código fuente y reiniciar automáticamente el servidor de aplicaciones Node.js cuando detecta un cambio.
- Para instalarlo:
  - npm install -g nodemon
- Para ejecutar el servidor en modo monitorizado:
  - nodemon server.js

Node REPL

- Un bucle Lectura-Evaluación-Impresión ("REPL" o "Read-Eval-Print-Loop"), también conocido como alto nivel interactivo o consola de lenguaje, es un entorno de programación computacional simple e interactivo que toma las entradas individuales del usuario, las evalúa y devuelve el resultado al usuario; un programa escrito en un entorno REPL es ejecutado parte por parte.
- Node Read-Eval-Print-Loop (REPL) está disponible como un programa independiente y fácilmente puede incluirse en otros programas. REPL proporciona una forma interactiva de ejecutar JavaScript y ver los resultados. Puede ser utilizado para la depuración, testing o simplemente para probar cosas.
- Para acceder se debe ejectuar node sin ningún argumento desde la línea de comandos, debe posicionarse dentro de REPL. Posee la edición simple de líneas de emacs.

```
$ node
Welcome to Node.js v10.13.0.
Type ".help" for more information.
> console.log('Hola mundo');
Hola mundo
undefined
> .exit
```

Yeoman: Generador del esqueleto web

- http://Yeoman.io
- Instalación:
  - npm install -g yo
  - npm install -g generator-angular
  - npm install -g generator-karma
- Generar un servidor y sitio web:
  - yo angular
- Descargar dependencias si es necesario
  - bower install & npm install
- Preparar entorno de ejecución y levantar el servidor en modo prueba:
  - grunt serve

Express: Infraestructura de aplicaciones web Node.js

- Instalación:
  - npm install -g express-generator
- Generar un servidor y sitio web:
  - express cursoserver
- Descargar dependencias
  - cd cursoserver && npm install
- Levantar servidor:
  - SET DEBUG=cursoserver:* & npm start
- Directorio de cliente de la aplicación web
  - cd cursoserver\public ← Copiar app. web

Grunt: Automatización de tareas

- Instalación general:
  - npm install -g grunt-cli
  - npm install -g grunt-init
- Instalación local de los módulos en la aplicación (añadir al fichero package.json de directorio de la aplicación o crearlo si no existe):

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
    "grunt-shell": "~0.7.0"
  }
}
```


Grunt: Automatización de tareas

- Descargar e instalar localmente los módulos en la aplicación (situarse en el directorio de la aplicación)
  - npm install --save-dev
- Fichero de tareas: gruntfile.js
- Ejecutar tareas:
  - grunt
  - grunt jshint
  - grunt reléase
  - grunt serve

Karma: Gestor de Pruebas unitarias de JavaScript

- Instalación general:
  - npm install -g karma
  - npm install -g karma-cli
- Generar fichero de configuración karma.conf.js:
  - karma init
- Ingenierías de pruebas unitarias disponibles:
  - http://jasmine.github.io/
  - http://qunitjs.com/
  - http://mochajs.org
  - https://github.com/caolan/nodeunit
  - https://github.com/nealxyc/nunit.js

Node Package Manager

NPM

¿Cómo usamos componentes externos?

1. Localizar la página
2. Descargar la versión más reciente
3. Descomprimir
4. Añadir referencias
5. Leer documentación
6. Modificar configuración
7. ¿Hay dependencias?

npm: Node Package Manager

- Npm es el administrador de paquetes estándar para Node.js. Cuenta con un repositorio donde se encuentran registrados más de 350.000 paquetes (2017) con la confianza de más de 11 millones de desarrolladores en todo el mundo, lo que lo convierte en el repositorio de código de un solo lenguaje más grande del mundo, y se puede estar seguro de que hay un paquete para (¡casi!) todo.
- Npm simplifica el uso de las dependencias externas permitiedo:
  - Localización
  - Descarga (¡con dependencias!)
  - Instalación / desinstalación
  - Configuración
  - Actualización
- Comenzó en 2009 como una forma de descargar y administrar las dependencias de los paquetes de Node.js, pero desde entonces se ha ampliado a dependencias de aplicaciones web front-end, aplicaciones móviles, robots, enrutadores y muchas otras necesidades de la comunidad de JavaScript.

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
  - npm update -g npm
- Verificar desde consola de comandos:
  - npm --version
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

package.json

- El archivo package.json es una especie de manifiesto del proyecto. Puede hacer muchas cosas: es un repositorio central de configuración de herramientas, es donde npm y yarn almacenan los nombres y versiones de todos los paquetes instalados, permite definir tareas de ejecución, …
- Para generar el fichero package.json:
  - npm init
- La estructura del archivo puede contener:
  - name: establece el nombre de la aplicación/paquete
  - version: indica la versión actual
  - description: es una breve descripción de la aplicación/paquete
  - main: establece el punto de entrada para la aplicación node
  - private: si se configura para true evita que la aplicación/paquete se publique accidentalmente en npm
  - scripts: define un conjunto de scripts que puede ejecutar npm
  - dependencies, devDependencies, optionalDependencies: establecen la lista de paquetes npm instalados como dependencias

package.json

- El archivo puede contener información adicional para su publicación en el repositorio de nmp:
  - author: muestra los datos del autor del paquete
    "author": "Joe <joe@whatever.com> (https://whatever.com)"
  - contributors: además del autor, el proyecto puede tener uno o más colaboradores. Esta propiedad es una matriz que los enumera.
  - bugs: enlace al rastreador de problemas del paquete, muy probablemente una página de problemas de GitHub
  - homepage: establece la url del sitio del paquete
  - license: Indica la licencia del paquete.
  - keywords: contiene una matriz de palabras clave que se asocian con lo que hace el paquete para facilitar su busqueda en el repositorio de npm.
  - repository: especifica dónde se encuentra este repositorio con los fuentes del paquete.
    "repository": "github:whatever/testing",
  - engines: establece en qué versiones de Node.js funciona este paquete/aplicación
    "engines": { "node": ">= 6.0.0", "npm": ">= 3.0.0", "yarn": "^0.13.0" }
  - browserslist: se utiliza para indicar qué navegadores (y sus versiones) desea admitir. Si se desea admitir las últimas 2 versiones principales de todos los navegadores con al menos un 1% de uso (de las estadísticas de CanIUse.com), excepto IE8 y versiones anteriores.
    "browserslist": ["> 1%", "last 2 versions", "not ie <= 8"]

Dependencias

- Las dependencias son los recursos externos o de terceros de los que depende el proyecto para su correcto funcionamiento y que no se han desarrollado dentro del propio proyecto.
- Se pueden realizar dos tipos de instalación:
  - Global: comandos y dependencias compartidas por múltiples proyectos:
    - npm install -g *lista de paquetes*
    - npm root -g (indica dónde la ubicación exacta en la máquina)
  - Local (predeterminada): comandos y dependencias propias del proyecto.
    - se instala en el árbol de archivos actual, en la subcarpeta node_modules.
    - Si el proyecto tiene un archivo package.json, se registra en el.

Dependencias

- Las dependencias se clasifican en:
  - dependencies: se incluyen con la aplicación en producción
  - devDependencies: contienen las herramientas de desarrollo
  - optionalDependencies: la falta de la dependencia no hará que la instalación falle pero es responsabilidad del programa manejar la falta de dependencias
- La clasificación se establece con las opciones de instalación:
  --save (-S): instala y la registra en dependencies del package.json (predeterminada desde npm 5)
  --save-dev (-D): instala y la registra en devDependencies del package.json
  --save-optional (-O): instala y la registra en optionalDependencies del package.json
  --no-save: instala pero no la registra en package.json
  npm install --sabe-dev *lista de paquetes*
- Para instalar todas las dependencias del package.json o solo las no opcionales:
  - npm install
  - npm install --no-optional

Versionado

- Además de las descargas simples, npm también administra el control de versiones , por lo que se puede especificar cualquier versión específica de un paquete o solicitar una versión superior o inferior a la que necesita.
- Se puede instalar una versión anterior de un paquete npm usando la sintaxis @:
  - npm install *package@versión*
  - npm install *package@latest*
- Para ver la versión de todos los paquetes npm instalados, incluidas sus dependencias:
  - npm list
- Para ver todas las versiones disponibles de un paquete:
  - npm view *package* versions

SEMVER

- El sistema SEMVER (Semantic Versioning) es un conjunto de reglas para proporcionar un significado claro y definido a las versiones de los proyectos de software.
- El sistema SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z, donde:
  - X se denomina Major: indica cambios rupturistas
  - Y se denomina Minor: indica cambios compatibles con la versión anterior
  - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.

Reglas de actualización

- ^: solo realizará actualizaciones que no cambien el número distinto de cero que se encuentra más a la izquierda, es decir, puede haber cambios en la versión secundaria o en la versión del parche, pero no en la versión principal. Si escribe ^13.1.0, al ejecutar npm update, puede actualizarse a 13.2.0, 13.3.0even 13.3.1, 13.3.2 y así sucesivamente, pero no a 14.0.0 o superior.
- ~: si se escribe ~0.13.0 al ejecutar npm update se permite actualizarse a las versiones de parches: 0.13.1 está permitido, pero 0.14.0 no lo está.
- >: se permite cualquier versión superior a la especificada
- >=: se permite cualquier versión igual o superior a la especificada
- <=: se permite cualquier versión igual o inferior a la especificada
- <: se permite cualquier versión inferior a la especificada
- =: solo se acepta esa versión exacta
- -: se permite un rango de de versiones. Ejemplo: 2.1.0 - 2.6.2
- ||: combinas conjuntos. Ejemplo: < 2.1 || > 2.6

Actualizaciones

- Para descubrir nuevos lanzamientos de paquetes:
  - npm outdated
- Para actualizar a una nueva versión menor o un parche:
  - npm update
  - npm update *package*
- Para actualizar a una nueva mayor:
  - npm install *package*
- Aprovechando npm-check-updates, se pueden actualizar todas las dependencias package.json a la última versión para luego hacer la intalación.
  - npm install -g npm-check-updates
  - ncu –u
  - npm install
- Para desinstalar un paquete:
  - npm uninstall *package*

package-lock.json

- En la versión 5, npm introdujo el archivo package-lock.json.
- El objetivo del package-lock.json es realizar un seguimiento de la versión exacta de cada paquete que se instala para que un producto sea 100% reproducible de la misma manera, incluso si los mantenedores actualizan los paquetes.
- Esto resuelve un problema muy específico que package.json dejó sin resolver. En package.json puede establecer a qué versiones se desea actualizar (parche o menor), utilizando la notación semver. Esto permitia que el proyecto original y la versión del proyecto recién descargada recibiera diferentes versiones de las dependencias al realizar el npm install lo que puede provocar problemas.
- package-lock.json establece "en piedra" la versión instalada actualmente de cada paquete, y npm usará esas versiones exactas cuando se ejecute npm ci o npm install-clean.

Tareas de ejecución

- El archivo package.json admite un formato para especificar tareas de línea de comandos que se pueden ejecutar mediante npm run.
- La propiedad scripts del package.json permite definir pares: nombre de comando:instrucción de línea de comandos:

  "debug": "node --inspect server-app"
- Las acciones de los comandos de npm start, restart, stop y test se definen en la propiedad scripts del package.json, cullos nombres de comandos coinciden. También se pueden definir los eventos previos y posteriores a dichos comandos que se ejecutan automáticamente, los nombres deben ser: prestart, poststart, prerestart, postrestart, prestop, poststop.

JavaScript 6: http://www.ecma-international.org/ecma-262/6.0/

ECMASCRIPT 2015

ECMAScript 2015 (ES6) y más allá

- Node.js está construido en base a versiones modernas de V8. Las últimas versiones del motor aseguran las nuevas características de la Especificación ECMA-262 de JavaScript a los desarrolladores de Node.js, así como mejoras continuas en el rendimiento y la estabilidad.
- Todas las características de ECMAScript 2015 (ES6) se dividen en tres grupos shipping, staged y in progress:
  - Todas las funcionalidades en shipping, que V8 considera estable, se activan de forma predeterminada en Node.js y hacen que NO se requiera ninguna bandera o flag en tiempo de ejecución.
  - Las funciones en staged, que son características casi completas que el equipo V8 no las considera estables, requieren un bandera o flag en tiempo de ejecución: --harmony.
  - In progress, las características pueden ser activadas individualmente por su respectiva bandera o flag, aunque esto es altamente desaconsejado a menos que sea para propósitos de pruebas.

https://node.green/

Introducción

- En Junio de 2015 aparece la 6ª edición del estándar ECMAScript (ES6), cuyo nombre oficial es ECMAScript 2015.
- Se está extendiendo progresivamente el soporte a ES6, pero aún queda trabajo por hacer.
- Las alternativas para empezar a utilizarlo son:
  - Transpiladores ("Transpiler": "Tanslator" y "Compiler"): Traducen o compilan un lenguaje de alto nivel a otro lenguaje de alto nivel, en este caso código ES6 a ES5. Los más conocidos y usados son Babel, TypeScript, Google Traceur, CoffeeScript.
  - Polyfill: Un trozo de código o un plugin que permite tener las nuevas funcionalidades de HTML5 o ES6 en aquellos navegadores que nativamente no lo soportan.

Declaración de variables

- let: ámbito de bloque, solo accesible en el bloque donde está declarada.

```
(function() {
   if(true) {
      let x = "variable local";
   }
   console.log(x); // error, "x" definida dentro del "if"
})();
```

- const: constante, se asina valor al declararla y ya no puede cambiar de valor.

```
const PI = 3.15;
PI = 3.14159; // error, es de sólo-lectura
```


Destructuring

- Asignar (repartir) los valores de un objeto o array en varias variables:

```
var [a, b] = ["hola", "adiós"];
console.log(a); // "hola"
console.log(b); // "adiós"

var obj = { nombre: "Pepito", apellido: "Grillo" };
var { nombre, apellido } = obj;
console.log(nombre); // "Pepito"
```

Template Strings

- Interpolación: sustituye dentro de la cadena las variable por su valor:

```
let var1 = "JavaScript";
let var2 = "Templates";
console.log(`El ${var1} ya tiene ${var2}.`);
// El JavaScript ya tiene Templates.
```

- Las constantes cadenas pueden ser multilínea sin necesidad de concatenarlos con +.

```
let var1 = " El JavaScript
ya tiene
Templates ";
```

- Incorpora soporte extendido uso de Unicode en cadenas y expresiones regulares.

Parámetros de funciones

- Valores por defecto: Se pueden definir valores por defecto a los parámetros en las funciones.

```
function(valor = "foo") {...};
```

- Resto de los parámetros: Convierte una lista de parámetros en un array.

```
function f (x, y, ...a) {
   return (x + y) * a.length
}
f(1, 2, "hello", true, 7) === 9
```

- Operador de propagación: Convierte un array o cadena en una lista de parámetros.

```
var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
```

Función Arrow

- Funciones anónimas.

```
data.forEach(elem => {
  console.log(elem);
  // …
});
var fn = (num1, num2) => num1 + num2;
pairs = evens.map(v => ({ even: v, odd: v + 1 }));
```

- this: dentro duna función Arrow hace referencia al contenedor y no al contexto de la propia función.

```
bar : function() {
    document.addEventListener("click", (e) => this.foo());
  }
```

- equivale a (ES5):

```
bar : function() {
    document.addEventListener("click", function(e) {
      this.foo();
    }.bind(this));
  }
```


Clases

- Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objetos que realizábamos en el estándar anterior, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como por ejemplo, herencia.

```
class LibroTecnico extends Libro {
  constructor(tematica, paginas) {
    super(tematica, paginas);
    this.capitulos = [];
    this.precio = "";
    // ...
  }
  metodo() {
    // ...
  }
}
```

Static Members

```
class Rectangle extends Shape {
   …
   static defaultRectangle () {
      return new Rectangle("default", 0, 0, 100, 100)
   }
}
class Circle extends Shape {
   …
   static defaultCircle () {
      return new Circle("default", 0, 0, 100)
   }
}
var defRectangle = Rectangle.defaultRectangle()
var defCircle    = Circle.defaultCircle()
```


Getter/Setter

```
class Rectangle {
   constructor (width, height) {
      this._width  = width
      this._height = height
   }
   set width  (width)  { this._width = width            }
   get width  ()       { return this._width             }
   set height (height) { this._height = height          }
   get height ()       { return this._height            }
   get area   ()       { return this._width * this._height }
}
var r = new Rectangle(50, 20)
r.area === 1000
```

mixin

- Soporte para la herencia de estilo mixin mediante la ampliación de las expresiones que producen objetos de función.

```
var aggregation = (baseClass, ...mixins) => {
  let base = class _Combined extends baseClass {
    constructor (...args) {
      super(...args)
      mixins.forEach((mixin) => {
        mixin.prototype.initializer.call(this)
      })
    }
  }
  let copyProps = (target, source) => {
    Object.getOwnPropertyNames(source)
      .concat(Object.getOwnPropertySymbols(source))
      .forEach((prop) => {
      if (prop.match(/^(?:constructor|prototype|arguments|caller|name|bind|call|apply|toString|length)$/))
        return
      Object.defineProperty(target, prop, Object.getOwnPropertyDescriptor(source, prop))
    })
  }
  mixins.forEach((mixin) => {
    copyProps(base.prototype, mixin.prototype)
    copyProps(base, mixin)
  })
  return base
}
```


Módulos

- Estructura el código en módulos similares a los espacios de nombres
- Llamamos a las funciones desde los propios Scripts, sin tener que importarlos en el HTML, si usamos JavaScript en el navegador.

```
//File: lib/person.js
module "person" {
  export function hello(nombre) {
    return nombre;
  }
}
```

Y para importar en otro fichero:

```
//File: app.js
import { hello } from "lib/person";
var app = {
  foo: function() {
    hello("Carlos");
  }
}
export app;
```

Módulos

- Ficheros como módulos:
  - Solo se puede importar lo previamente exportado.
  - Es necesario importar antes de utilizar
  - El fichero en el from sin extensión y ruta relativa (./ ../) o sin ruta (NODE_MODULES)
- Exportar:

```
export public class MyClass { }
export { MY_CONST, myFunction, name as otherName }
```

- Importar:

```
import * from './my_module';
import * as MyModule from './my_module';
import { MyClass, MY_CONST, myFunction as func } from './my_module';
```

- Pasarelas: Importar y exportar (index.ts)

```
export { MyClass, MY_CONST, myFunction as func } from './my_module';
```


Iteradores y Generadores

- El patrón Iterador permite trabajar con colecciones por medio de abstracciones de alto nivel
- Un Iterador es un objeto que sabe como acceder a los elementos de una secuencia, uno cada vez, mientras que mantiene la referencia a su posición actual en la secuencia.
- En ES2015 las colecciones (arrays, maps, sets) son objetos iteradores.
- Los generadores permiten la implementación del patrón Iterador.
- Los Generadores son funciones que pueden ser detenidas y reanudadas en otro momento.
- Estas pausas en realidad ceden la ejecución al resto del programa, es decir no bloquean la ejecución.
- Los Generadores devuelven (generan) un objeto "Iterator" (iterador)

Generadores

- Para crear una función generadora

```
function* myGenerator() {
  // ...
  yield value;
  // ...
}
```

- La instrucción yield devuelve el valor y queda a la espera de continuar cuando se solicite el siguiente valor.
- El método next() ejecuta el generador hasta el siguiente yield dentro del mismo y devuelve un objeto con el valor.

```
var iter = myGenerator();
// ...
rslt = iter.next();
// ...
```

- La nueva sintaxis del for permite recorrer el iterador completo:

```
for (let value of iter)
```


Nuevos Objetos

- Map: Lista de pares clave-valor.
- Set: Colección de valores únicos que pueden ser de cualquier tipo.
- WeakMap: Colección de pares clave-valor en los que cada clave es una referencia de objeto.
- WeakSet: Colección de objetos únicos.
- Promise: Proporciona un mecanismo para programar el trabajo de modo que se lleve a cabo en un valor que todavía no se calculó.
- Proxy: Habilita el comportamiento personalizado de un objeto.
- Reflect: Proporciona métodos para su uso en las operaciones que se interceptan.
- Symbol: Permite crear un identificador único.
- Intl.Collator: Proporciona comparaciones de cadenas de configuración regional.
- Intl.DateTimeFormat: Proporciona formato de fecha y hora específico de la configuración regional.
- Intl.NumberFormat: Proporciona formato de número específico de la configuración regional.

Nuevos Objetos

- ArrayBuffer: Representa un búfer sin formato de datos binarios, que se usa para almacenar datos de las diferentes matrices con tipo No se puede leer directamente de Arraybuffer ni escribir directamente en Arraybuffer, pero se puede pasar a una matriz con tipo o un objeto DataView para interpretar el búfer sin formato según sea necesario.
- DataView: Se usa para leer y escribir diferentes tipos de datos binarios en cualquier ubicación de ArrayBuffer.
- Float32Array: Matriz con tipo de valores flotantes de 32 bits.
- Float64Array: Matriz con tipo de valores flotantes de 64 bits.
- Int8Array: Matriz con tipo de valores enteros de 8 bits.
- Int16Array: Matriz con tipo de valores enteros de 16 bits.
- Int32Array: Matriz con tipo de valores enteros de 32 bits.
- Uint8Array: Matriz con tipo de valores enteros sin signo de 8 bits.
- Uint8ClampedArray: Matriz con tipo de enteros sin signo de 8 bits con valores fijos.
- Uint16Array: Matriz con tipo de valores enteros sin signo de 16 bits.
- Uint32Array: Matriz con tipo de valores enteros sin signo de 32 bits.

Promise Pattern

- El Promise Pattern es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutaran a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón Callback:
  - Llamadas anidadas
  - Complejidad de código

  o.m(1, 2, f(m1(3, f1(4,5,ff(8))) → o.m(1, 2).f().m1(3).f1(4 , 5).ff(8)
- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- El servicio $q es un servicio de AngularJS que contiene toda la funcionalidad de las promesas (está basado en la implementación Q de Kris Kowal).
- La librería JQuery incluye el objeto $.Deferred desde la versión 1.5.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión 6.

Objeto Promise

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, …) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
  - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
  - Resuelta: Se ha podido obtener el resultado (Promise.resolve())
  - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (Promise.reject())
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

Crear promesas

- El objeto Promise gestiona la creación de la promesa y los cambios de estados de la misma.

```
list() {
  return new Promise((resolve, reject) => {
   this.http.get(this.baseUrl).subscribe(
       data => resolve(data),
       err => reject(err)
       )
  });
 }
```

- Para crear promesas ya concluidas:
  - Promise.reject: Crea una promesa nueva como rechazada cuyo resultado es igual que el argumento pasado.
  - Promise.resolve: Crea una promesa nueva como resuelta cuyo resultado es igual que su argumento.

Invocar promesas

- El objeto Promise creado expone los métodos:
  - then(fnResuelta, fnRechazada): Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
  - catch(fnError): Recibe como parámetro la función a ejecutar en caso de que falle.

    list().then(calcular, ponError).then(guardar)
- Otras formas de crear e invocar promesas son:
  - Promise.all: Combina dos o más promesas y realiza la devolución solo cuando todas las promesas especificadas se completan o alguna se rechaza.
  - Promise.race: Crea una nueva promesa que resolverá o rechazará con el mismo valor de resultado que la primera promesa que se va resolver o rechazar entre los argumentos pasados.

asyc/await (ES2017)

- La declaración de función asincrónica define una función asincrónica, que devuelve un objeto AsyncFunction. Una función asincrónica es una función que opera asincrónicamente a través del bucle de eventos, utilizando una promesa implícita para devolver su resultado. Pero la sintaxis y la estructura de su código usando funciones asíncronas se parece mucho más a las funciones síncronas estándar.
- El operador await se usa para esperar a una Promise y sólo dentro de una async function.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => { setTimeout(() => { resolve(x); }, 2000); });
}
async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}
```

CONCEPTOS

Creación de una aplicación con node.js

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hola mundo.\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Creación de una aplicación con node.js

- Para crear una aplicación típica con Node.js debemos tener en cuenta los sientes tres pasos importantes:
  - Importación de módulos necesarios
  - Crear controladores de eventos
  - Asociar los controladores de eventos.
- Para ejecutar la aplicación:
  - node hola-mundo.js
- Para acceder a la aplicación:
  - http://127.0.0.1:3000/

Depuración

- Cuando se inicia con la opción --inspect, un proceso Node.js escuchara a un cliente de depuración. Por defecto, escuchará en el host y el puerto 127.0.0.1:9229. A cada proceso también se le asigna un UUID único.
  - node --inspect hola-mundo.js
- Los clientes inspectores deben conocer y especificar la dirección del host, el puerto y el UUID para conectarse. Una URL completa se verá algo así ws://127.0.0.1:9229/5e6a1c52-626d-48ba-93c2-0d3b8978868b.
- Chrome DevTools 55+
  - Abrir chrome://inspect en un navegador basado en Chromium o edge://inspect en Edge. Haga clic en el botón Configurar y asegúrese de que su host y puerto de destino estén en la lista.
- Visual Studio Code 1.10+
  - En el panel Depurar, haga clic en el icono de configuración para abrir .vscode/launch.json. Seleccione "Node.js" para la configuración inicial.

Objetos Globales

- global:
  - En los navegadores, el alcance de nivel superior es el alcance global. Esto significa que dentro del navegador var algo definirá una nueva variable global. En Node.js esto es diferente. El alcance de nivel superior no es el alcance global; var algo dentro de un módulo Node.js será local para ese módulo.
  - El objeto global expone los métodos temporizadores.
- module:
  - Representa el módulo actual que expone como global __dirname, __filename, exports, module, require(id)
- process:
  - Proporciona información, eventos y control sobre el proceso actual de Node.js.
- console:
  - Proporciona una consola de depuración simple que es similar al mecanismo de consola de JavaScript proporcionado por los navegadores web.

process

- El módulo principal process de Node.js proporciona la propiedad env que aloja todas las variables de entorno que se establecieron en el momento en que se inició el proceso.
- Para establecer las variables de entorno en el sistema operativo:

  C:\> set PORT=8080

  $ PORT=8080
- Para acceder a las variables de entorno desde Node:

  console.log(process.env.PORT || '8080')
- Si hay múltiples variables de entorno en el proyecto, también se puede crear un archivo .env (NOMBRE=VALOR) en el directorio raíz del proyecto y usar el paquete dotenv para cargarlas durante el tiempo de ejecución.
  - npm install dotenv
- Para cargar los valores desde línea de comandos:
  - node -r dotenv/config app.js
- Se pueden carga las variables de entorno desde código con:

  require('dotenv').config();

process

- Como alternativa a las variables de entorno, se puede pasar cualquier cantidad de argumentos al invocar una aplicación. Los argumentos pueden ser independientes o tener una clave y un valor.
- El objeto process expone la propiedad argv con una matriz que contiene todos los argumentos de invocación de la línea de comandos.
  - El primer elemento es la ruta completa del comando.
  - El segundo elemento es la ruta completa del archivo que se está ejecutando.
  - Todos los argumentos adicionales están presentes desde la tercera posición en adelante.
- Para el comando:
  - node app.js debug PORT=4444
- Se pueden recuperar los argumentos con:

```
process.argv.slice(2).forEach((val, index) => {
   console.log(`${index}: ${val.includes('=') ? `name: ${val.split('=')[0]} value:
         ${val.split('=')[1]}` : val}`);
});
```


Salida básica usando la consola

- Node.js proporciona un módulo console que ofrece muchas formas muy útiles de interactuar con la línea de comandos. Es básicamente lo mismo que el objeto console que encuentras en el navegador.
- El método más básico y más utilizado es console.log(), que imprime la cadena que le pasas a la consola. Si se le pasa un objeto, lo representará como una cadena y se pueden pasar múltiples variables.

```
console.log('Hola mundo')
```

- También permite formatear frases pasando un especificador de formato y variables.
  - %s formatear una variable como una cadena
  - %d dar formato a una variable como un número
  - %i formatear una variable solo con su parte entera
  - %o dar formato a una variable como un objeto

```
console.log('Mi %s tiene %d patas' , 'gato' , 4)
```

- console.clear() borra la consola (el comportamiento puede depender de la consola utilizada)

Entrada desde la línea de comando

- Node.js, desde la versión 7, proporciona el módulo readline para obtener información de un flujo legible como el flujo process.stdin, que durante la ejecución de un programa Node.js es la entrada del terminal, una línea a la vez.

```
const readline = require('readline').createInterface({
    input: process.stdin, output: process.stdout,
});
readline.question(`¿Como te llamas? `, name => {
    console.log(`Hola ${name}`);
    readline.close();
});
```


Módulos

- Node posee un sencillo sistema de carga basado en CommonJS (los módulos ES2015 se incorporaron posteriormente). En Node, los ficheros y módulos son de correspondencia biunívoca.
- Un módulo es un fichero que exporta parcial o totalmente su contenido.
- El fichero circle.js:

```
var PI = Math.PI;
exports.area = function (r) { return PI * r * r; };
exports.circumference = function (r) {  return 2 * PI * r; };
```

- El módulo circle.js ha exportado las funciones area() y circumference().
- Las variables locales y funciones del módulo serán privadas. En este ejemplo la variable PI es privada en circle.js.
- Para exportar a un objeto, debe añadir el objeto especial exports.

```
module.exports = class Square {
  constructor(width) { this.width = width;  }
  area() { return this.width ** 2; }
};
```

Módulos

- La carga de los módulos se realiza con la directiva required:

```
var circle = require('./circle.js');
console.log( 'El área de un círculo con radio 4 es '
        + circle.area(4));
```

- Los módulos en Node.js no se inyectan automáticamente en el ámbito global, sino que se asignan a una variable de libre elección. Los elementos exportados son accesibles como miembros de la variable por lo que no hay que preocuparse por dos o más módulos que tienen funciones con el mismo nombre.
- Node posee varios módulos básicos compilados en binario. Los módulos básicos están definidos en el código fuente de node en la carpeta lib/.
- Los módulos básicos tienen la preferencia de cargarse primero si su identificador es pasado desde require(), incluso si hay un fichero con ese nombre.

Módulos

- Si el nombre exacto del fichero no es encontrado, entonces node intentará cargar el nombre del fichero seguido de la extensión .js, y a continuación con .node.
- Los ficheros .js son interpretados como ficheros de texto en JavaScript, y los ficheros .node son interpretados como extensiones de módulos compilados cargados con dlopen.
- Un módulo con el prefijo '/' indica la ruta absoluta al fichero.
- Un módulo con el prefijo './' o '../' es relativo al fichero que invoca el require().
- Si se omite el uso de '/' o './' en el fichero, el módulo puede ser un "módulo básico" o se cargará desde la carpeta node_modules.
- Es conveniente organizar los programas y librerías en los mismos directorios, y proporcionar un único punto de entrar a la biblioteca.
- Se puede crear el fichero package.json en la raíz de la carpeta, que especifique el módulo main:

```
{ "name" : "some-library", "main" : "./lib/some-library.js" }
```

- Si ./some-library es una carpeta, entonces require('./some-library') trataría de cargar ./some-library/lib/some-library.js.
- Si no hay ningún fichero package.json presente en el directorio, entonces node intentará cargar el fichero index.js o index.node de ese directorio.

Módulos

- Los módulos se almacenan en caché después de la primera vez que se cargan. Esto significa (entre otras cosas) que cada llamada a require('foo') obtendrá exactamente el mismo objeto devuelto, si se resuelve en el mismo archivo.
- Siempre que require.cache no se modifique, varias llamadas a require('foo') no harán que el código del módulo se ejecute varias veces. Esta es una característica importante. Con él, se pueden devolver objetos "parcialmente hechos", lo que permite cargar dependencias transitivas incluso cuando causen ciclos.
- Los módulos se almacenan en caché según su nombre de archivo resuelto. Dado que los módulos pueden resolverse con un nombre de archivo diferente según la ubicación del módulo de llamada (cargando desde carpetas node_modules), no está garantizado que require('foo') siempre devolverá exactamente el mismo objeto, si se resolviera en archivos diferentes.

Bloqueo frente a no bloqueo

- El bloqueo es cuando la ejecución de JavaScript adicional en el proceso Node.js debe esperar hasta que se complete una operación que no sea JavaScript. Esto sucede porque el bucle de eventos no puede continuar ejecutando JavaScript mientras se produce una operación de bloqueo.
- Los métodos de bloqueo se ejecutan sincrónicamente y los métodos sin bloqueo se ejecutan asincrónicamente .
- Usando el módulo del sistema de archivos como ejemplo, este es un archivo leído síncronamente:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

- El ejemplo asíncrono equivalente :

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

Bloqueo frente a no bloqueo

- La ejecución de JavaScript en Node.js es de un solo subproceso, por lo que la concurrencia se refiere a la capacidad del bucle de eventos para ejecutar funciones de devolución de llamada (Callbacks) de JavaScript después de completar otro trabajo.
- Cualquier código que se espera que se ejecute de manera concurrente debe permitir que el bucle de eventos continúe ejecutándose a medida que se producen operaciones que no son JavaScript, como E / S. Todas las API de Node se escriben de tal manera que admiten Callbacks.
- El bucle de eventos es diferente a los modelos en muchos otros lenguajes donde se pueden crear hilos adicionales para manejar el trabajo concurrente.

Bloqueo frente a no bloqueo

- El bucle de eventos es lo que permite a Node.js realizar operaciones de E / S sin bloqueo, a pesar del hecho de que JavaScript es de un solo subproceso, descargando operaciones al núcleo del sistema siempre que sea posible.
- Dado que la mayoría de los núcleos modernos son multiproceso, pueden manejar múltiples operaciones que se ejecutan en segundo plano. Cuando se completa una de estas operaciones, el núcleo le dice a Node.js que se puede agregar la devolución de llamada apropiada a la cola de sondeo para que finalmente se ejecute.
- Cuando se inicia Node.js, inicializa el bucle de eventos, procesa el script de entrada proporcionado (que puede realizar llamadas API asíncronas, programar temporizadores o llamadas process.nextTick()) y comienza a procesar el bucle de eventos.
- Cuando las diferentes operaciones van terminando, ingresan su callback en la cola de eventos que los ejecuta secuencialmente.

Bloqueo frente a no bloqueo

- El bucle de eventos está dividido en diferentes fase:
  - temporizadores: esta fase ejecuta devoluciones de llamada programadas por setTimeout() y setInterval().
  - devoluciones de llamada pendientes: ejecuta devoluciones de llamada de E / S diferidas a la siguiente iteración de bucle.
  - inactivo, preparar: solo se usa internamente.
  - encuesta: recuperar nuevos eventos de E / S; ejecutar devoluciones de llamada relacionadas con E / S (casi todas con la excepción de devoluciones de llamada cercanas, las programadas por temporizadores y setImmediate()). Node se bloqueará aquí cuando sea apropiado.
  - comprobar: aquí se invocan las devoluciones de llamada setImmediate().
  - devoluciones de llamada cercanas: algunas devoluciones de llamada cercanas, por ejemplo socket.on('close', ...).

Bloqueo frente a no bloqueo

- Cada fase tiene una cola FIFO de callbacks para ejecutar. Si bien cada fase es especial a su manera, generalmente, cuando el bucle de eventos ingresa a una fase determinada, realizará cualquier operación específica de esa fase, luego ejecutará callbacks en la cola de esa fase hasta que la cola se haya agotado o el número máximo de callbacks se ha completado o se alcanza el límite de devolución de llamada, el bucle de eventos se moverá a la siguiente fase, y así sucesivamente.
- Desde cualquiera de estas operaciones se pueden programar más operaciones y los nuevos eventos procesados en el sondeo de fase se ponen en cola por el núcleo, los acontecimientos de la encuesta pueden poner en cola mientras que los eventos electorales están siendo procesados. Como resultado, las devoluciones de llamada de larga duración pueden permitir que la fase de sondeo se ejecute mucho más tiempo que el umbral de un temporizador.

Temporizadores

- El módulo timer expone una API global para programar funciones que se llamarán en algún período de tiempo futuro. Debido a que las funciones del temporizador son globales, no hay necesidad de llamar require('timers') para usar la API.
- Las funciones de temporización dentro Node.js implementan un API similar a la API de temporizadores proporcionada por los navegadores web, pero utilizan una aplicación interna diferente que se construye alrededor de la Node.js Event Loop.
- Un temporizador en Node.js es una construcción interna que llama a una función dada después de un cierto período de tiempo. La llamada a la callback de un temporizador varía según el método utilizado para crear el temporizador y al trabajo que este haciendo el bucle de eventos Node.js.
- setImmediate: Programa la ejecución "inmediata" de las callback posteriores a los eventos de E / S.
  - Cuando se realizan varias llamadas a setImmediate(), las funciones callback se ponen en cola para su ejecución en el orden en que se crean. La cola de devoluciones de llamada se procesa completa en cada iteración del bucle de eventos. Si se pone en cola un temporizador inmediato desde el interior de un callback en ejecución, ese temporizador no se activará hasta la próxima iteración del bucle de eventos.
- setInterval: Programa la ejecución repetida de callback cada periodo en milisegundos.
- setTimeout: Programa la ejecución de una sola vez callback después de un periodo en milisegundos.
  - Es probable que el callback no se invoque precisamente el periodo exacto. Node.js no garantiza el momento exacto de cuándo se activarán las devoluciones de llamada, ni su orden. La devolución de llamada se llamará lo más cerca posible del tiempo especificado.
- Los métodos setImmediate(), setInterval() y setTimeout() devuelven objetos que representan los temporizadores y pueden ser usados para cancelar o evitar que se active el temporizador, con los métodos clearImmediate(), clearInterval() y clearTimeout().

Eventos

- Gran parte de la API central de Node.js se basa en una arquitectura asíncrona dirigida por eventos en la que ciertos tipos de objetos (llamados "emisores") emiten eventos con nombre que hacen que se invoquen objetos Function ("listeners" u "oyentes").
- Todos los objetos que emiten eventos son instancias de la clase EventEmitter. Estos objetos exponen una función eventEmitter.on() que permite adjuntar una o más funciones a los eventos con nombre emitidos por el objeto. Normalmente, los nombres de eventos son cadenas en notación Camell, pero se puede usar cualquier nombre de propiedad JavaScript válida.
- Cuando el objeto EventEmitter emite un evento, se invocan todas las funciones asociadas a ese evento específico de forma síncrona. Cualquier valor devuelto por los oyentes llamados se ignora y se descartará.
- El método eventEmitter.on() se utiliza para registrar oyentes, mientras que el método eventEmitter.emit() se utiliza para desencadenar el evento.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('event', () => console.log('an event occurred!'));
myEmitter.emit('event');
```

Eventos

- El método eventEmitter.emit() permite pasar un conjunto arbitrario de argumentos a las funciones de escucha. Es importante tener en cuenta que cuando se llama a una función de escucha ordinaria, la palabra clave estándar this se establece intencionalmente para hacer referencia a la instancia EventEmitter a la que está conectado el oyente.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this === myEmitter); // uno dos true
});
myEmitter.emit('event', 'uno', 'dos');
```

- Es posible utilizar las funciones de flecha ES6 como oyentes, sin embargo, al hacerlo, la palabra clave this ya no hará referencia a la instancia EventEmitter:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => console.log(a, b, this)); // uno dos {}
myEmitter.emit('event', 'uno', 'dos');
```


Eventos

- Se pueden eliminar oyentes concretos con eventEmitter.removeListener() o eventEmitter.off(), requieren la referencia al oyente.

```
const listener = item => console.log(`Valor: ${item}`);
myEmitter.on('event', listener);
// …
myEmitter.off('event', listener);
```

- Con eventEmitter.removeAllListeners() se eliminan todos los oyentes para el eventName especificado. Es una mala práctica eliminar los oyentes agregados en otras partes del código.
- Usando el método eventEmitter.once(), es posible registrar un oyente que se llama como máximo una vez para un evento en particular (lo elimina automáticamente).
- El objeto EventEmitter invoca a todos los oyentes síncronamente en el orden en que fueron registrados. Esto es importante para garantizar la secuencia adecuada de los eventos y para evitar condiciones de carrera o errores lógicos. Se puede cambiar a un modo de operación asíncrono utilizando los métodos setImmediate() o process.nextTick():

```
myEmitter.on('event', function(a, b) {
  setImmediate(() => { console.log(a, b); });
});
```

Streams

- Un flujo es una interfaz abstracta para trabajar con la transmisión (streaming) de datos en Node.js. Los Streams son objetos que permiten leer datos de un origen o escribir datos en un destino de manera continua.
- Node.js proporciona muchos objetos de flujo: una solicitud a un servidor HTTP o el process.stdout son instancias de flujo.
- El módulo stream proporciona una API para implementar la interfaz de flujos y es útil para crear nuevos tipos de instancias de transmisión, aunque por lo general, no es necesario usar el módulo stream para consumir flujos.

```
const stream = require('stream');
```

- En Node.js hay cuatro tipos de streams:
  - Readable: stream que se utiliza para la operación de lectura.
  - Writable: stream que se utiliza para la operación de escritura.
  - Duplex: stream que se puede utilizar para operaciones de lectura y escritura.
  - Transform: Tipo de stream duplex en el que la salida se calcula en función de la entrada.

Streams

- Cada tipo de Stream es una instancia de EventEmitter y lanza varios eventos en diferentes instancias de tiempos. Algunos de los eventos de uso común son:
  - Data: Este evento se activa cuando hay datos disponibles para leer.
  - End: Este evento se activa cuando no hay más datos que leer.
  - Error: Este evento se activa cuando hay algún error al recibir o escribir datos.
  - Finish: Este evento se activa cuando todos los datos han sido enviados al sistema subyacente.
- Al igual que con los tubos Unix, los flujos de Node implementan un operador de composición llamado .pipe(). Los principales beneficios del uso de streams son que no es que no es necesario almacenar todos los datos en la memoria y son fácilmente componibles.
- Por ejemplo se puede crear un flujo que lea un archivo, lo cifre usando el algoritmo AES-256, luego lo comprima usando gzip y termine escribiéndolo en otro fichero. Todo esto utilizando streams.

Streams

```
var crypto = require('crypto');
var fs = require('fs');
var zlib = require('zlib');

var password = Buffer.from(process.env.PASS || 'password');
var encryptStream = crypto.createCipher('aes-256-cbc', password);
var gzip = zlib.createGzip();
var readStream = fs.createReadStream(__filename);     // Este archivo
var writeStream = fs.createWriteStream(__dirname + '/out.gz');

readStream                      // Lee el archivo actual
  .pipe(encryptStream)          // Cifra
  .pipe(gzip)                   // Comprime
  .pipe(writeStream)            // Escribe en archivo
  .on('finish', function () {   // Hecho
   console.log('done');
});
```


Buffer

- Antes de la introducción de TypedArray, el lenguaje JavaScript no tenía mecanismo para leer o manipular flujos de datos binarios. La clase Buffer se introdujo como parte de la API Node.js para permitir la interacción con flujos de octetos en flujos TCP, operaciones del sistema de archivos y otros contextos.
- Con TypedArray ahora disponible, la clase Buffer implementa el Uint8ArrayAPI de una manera que es más optimizado y adecuado para Node.js.
- Las instancias de la clase Buffer son similares a las matrices de enteros de 0 a 255 pero corresponden a asignaciones de memoria sin procesar de tamaño fijo fuera del montón V8. El tamaño de la Buffer se establece cuando se crea y no se puede cambiar.
- La clase Buffer es una clase global a la que se puede acceder en una aplicación sin importar el módulo.

Buffer

- Para hacer que la creación de instancias de Buffer más fiables y menos propensas a errores, las diversas formas del constructor new Buffer() han sido marcadas como obsoletas.
- Para crear instancias de Buffer se debe utilizar:
  - Buffer.from(array): devuelve un nuevo Buffer que contiene una copia de los octetos proporcionados.
  - Buffer.from(arrayBuffer[, byteOffset[, length]]): devuelve un nuevo Buffer que comparte la misma memoria asignada que el ArrayBuffer dado.
  - Buffer.from(buffer): devuelve un nuevo Buffer que contiene una copia del contenido del Buffer dado.
  - Buffer.from(string[, encoding]): devuelve un nuevo Buffer que contiene una copia de la cadena proporcionada.
  - Buffer.alloc(size[, fill[, encoding]]): devuelve un nuevo Buffer inicializado del tamaño especificado. Este método es más lento Buffer.allocUnsafe(size) pero garantiza que las instancias recién creadas nunca contengan datos antiguos que sean potencialmente confidenciales.
  - Buffer.allocUnsafe(size) y Buffer.allocUnsafeSlow(size): cada uno devuelve un nuevo Buffer no inicializado del especificado size. Como Buffer no está inicializado, el segmento de memoria asignado puede contener datos antiguos que son potencialmente confidenciales.

Buffer

- Para crear el buffer:

```
var buf = Buffer.alloc(256, '');
```

- Para escribir en el buffer:

```
var len = buf.write('Hola mundo');
buf.writeInt8(77, 5);
buf[5]=109;
```

- Para recuperar el buffer:

```
console.log(buf.length, len);
console.log(buf.toString('utf8'));
console.log(buf.toJSON());
console.log(buf.readInt8(5), buf[5]);
```

- Dado que el buffer es un array permite las operaciones comunes de los mismos.
  - fill(), indexOf(), key(), values(), slice(), subarray(), …

Errores

- Las aplicaciones que se ejecutan en Node.js generalmente experimentarán cuatro categorías de errores:
  - Errores estándar de JavaScript como EvalError, SyntaxError, RangeError, ReferenceError, TypeError y URIError.
  - Errores del sistema provocados por restricciones subyacentes del sistema operativo, como intentar abrir un archivo que no existe o enviar datos a través de un socket cerrado.
  - Errores especificados por el usuario activados por el código de la aplicación.
  - Errores de aserciones son una clase especial de error que puede activarse cuando Node.js detecta una violación lógica excepcional que nunca debería ocurrir.
- Todos los errores de JavaScript y del sistema generados por Node.js heredan de la clase Error estándar de JavaScript y son una garantía de que proporcionan al menos las propiedades disponibles en esa clase.

Errores

- Node.js admite varios mecanismos para propagar y manejar errores que ocurren mientras se ejecuta una aplicación. La forma en que se informan y manejan estos errores depende completamente del tipo Error y el estilo de la API que se llama.
- Todos los errores de JavaScript se manejan como excepciones que generan y arrojan inmediatamente un error utilizando el throw estándar de JavaScript. Estos se manejan utilizando la construcción try…catch proporcionada por el lenguaje JavaScript.
- La instrucción throw generará una excepción que debe manejarse utilizando try…catch o el proceso Node.js se cerrará de inmediato.
- Con pocas excepciones, las API síncronas (cualquier método de bloqueo que no acepte una función callback, como fs.readFileSync), utilizarán throw para informar errores.
- Los errores que ocurren dentro de las API asincrónicas pueden informarse y tratarse de varias maneras.

Errores

- La mayoría de los métodos asíncronos expuestos por la API principal de Node.js siguen un patrón idiomático denominado error-first callback. Con este patrón, los métodos que aceptan una función callback definen como primer argumento un objeto Error. Si este primer argumento no es null y es una instancia de Error, entonces ha ocurrido un error que debería ser tratado.

```
fs.readFile('not exist file', (err, data) => {
  if (err) { … }
  // …
}
```

- El mecanismo try…catch de JavaScript no se puede utilizar para interceptar errores generados por API asíncronas.

```
try {
  fs.readFile('not exist file', (err, data) => { ... });
} catch (err) {
  // Todavía no se ha producido el error
}
```


Errores

- Cuando se llama a un método asincrónico en un objeto que es un EventEmitter o un Stream, los errores se pueden enrutar al evento 'error' del objeto.

```
myEmitter.on('error', (err) => { … });
```

- Los errores generados en los EventEmitter no se pueden interceptar utilizando try…catch ya que se generan después de que el código de llamada ya haya continuado.
- Los EventEmitter no deben utilizar la instrucción throw para lanzar excepciones, deben emitir un evento 'error':

```
myEmitter.emit('error', new Error('whoops!'));
```

- Para todos los objetos EventEmitter, si no se proporciona un controlador de eventos 'error', se generará un nuevo Error, causando que el proceso Node.js informe una excepción no controlada y se bloquee.
- Para evitar el bloqueo por excepciones no controladas se puede registrar un controlador para el evento 'uncaughtException' del proceso.

```
process.on('uncaughtException', (err, origin) => { … });
```

Promesas

- Para poder utilizar asyc/await es necesario convertir el estilo común de métodos que utilizan callback en promesas.
- Con el paso de las versiones Node ha ido añadiendo métodos homólogos a los existentes pero que devuelven promesas.
- El módulo util suministra el método promisify que toma un método que sigue el patrón error-first callback y genera la versión que devuelve promesas.

```
const util = require('util');
const fs = require('fs');

fs.readFile(__filename, (err, data) => {
  if (err) {
    // Handle the error.
  }
  // Do something with 'data'
});
const readFile = util.promisify(fs.stat);
readFile(__filename).then(data => {
  // Do something with 'data'
}).catch((err) => {
  // Handle the error.
});
async function callReadFile() {
  const data = await readFile(__filename);
  // Do something with 'data'
}
callReadFile();
```


Otros módulos

- Net
- HTTP
- HTTP/2
- HTTPS
- URL
- Query Strings
- DNS
- TLS/SSL
- UDP/Datagram
- Utilities
- String Decoder
- Timers
- Readline
- TTY
- File System
- Path
- Policies
- Web Streams API
- Crypto
- Zlib
- OS
- Cluster
- Worker Threads
- V8
- VM
- REPL

Hypertext Transfer Protocol

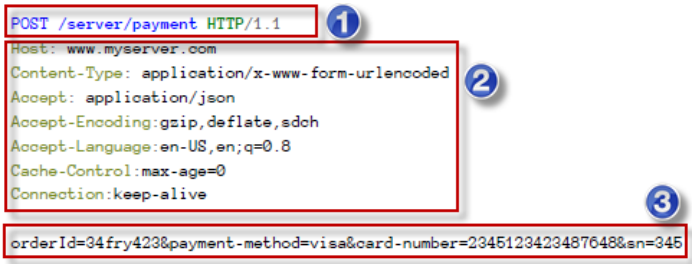
HTTP

HTTP

- HTTP es una pieza fundamental en World Wide Web, y especifica como intercambiar entre cliente y servidor recursos web.
- Características de HTTP:
  - Es un protocolo de nivel de aplicación y algo de presentación.
  - Está diseñado para ser ejecutado sobre TCP o sobre TLS/SSL.
  - Se basa en un paradigma sencillo de petición/respuesta, es decir, es un protocolo stateless.
  - Es un protocolo síncrono, la solicitud tiene que esperar la respuesta.

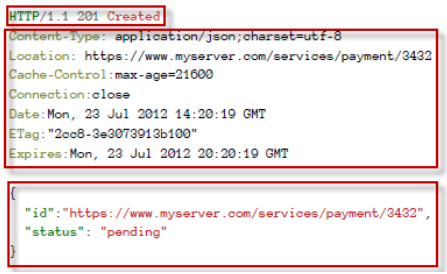
Petición HTTP

- Cuando realizamos una petición HTTP, el mensaje consta de:
  - Primera línea de texto indicando la versión del protocolo utilizado, el verbo y el URI
    - El verbo indica la acción a realizar sobre el recurso web localizado en la URI
  - Posteriormente vendrían las cabeceras (opcionales)
  - Después el cuerpo del mensaje, que contiene un documento, que puede estar en cualquier formato (XML, HTML, JSON → Content-type)



Respuesta HTTP

- Los mensajes HTTP de respuesta siguen el mismo formato que los de envío.
- Sólo difieren en la primera línea
  - Donde se indica un código de respuesta junto a una explicación textual de dicha respuesta.
  - El código de respuesta indica si la petición tuvo éxito o no.



Recursos

- Un recurso es cualquier elemento que dispone de un URI correcto y único.
- Es cualquier cosa que sea direccionable a través de internet.
- Estos recursos pueden ser manipulados por clientes y servidores.
  - Una noticia.
  - La temperatura en Madrid a las 22:00h.
  - Un estudiante de alguna clase en alguna escuela
  - Un ejemplar de un periódico, etc
- Todos los recursos tienen las mismas operaciones (CRUD)
  - CREATE, READ, UPDATE, DELETE

URI (Uniform Resource Identifier)

- Los URI son los identificadores globales de recursos en la web, y actúan de manera efectiva como UUIDs REST.
- Hay 2 tipos de URIs : URL y URN
  - URLs Identifican un recurso de red mediante una IP o un DNS
  - URNs son simples UUIDs lógicos con un espacio de nombres asociados
- URI es una cadena de caracteres corta, que identifica inequívocamente un recurso y que tienen el siguiente formato

  <esquema>://<host>:puerto/<ruta><querystring><fragmento>
  - Esquema: Indican que protocolo hay que utilizar para usar el recurso (http o https)
  - Host: Indica el lugar donde encontraremos el recurso (por IP o por dominio)
  - Puerto: Puerto por donde se establece la conexión (80 o 443)
  - Ruta: Ruta del recurso dentro del servidor, está separado por /
  - QueryStrng: Parámetros adicionales, separados por ? o por &
  - Fragmento: Separado por #

URI (Uniform Resource Identifier)

- Las URI es el único medio por el que los clientes y servidores pueden realizar el intercambio de representaciones.
- Normalmente estos recursos son accesibles en una red o sistema.
- Para que un URI sea correcto, debe de cumplir los requisitos de formato, REST no indica de forma específica un formato obligatorio.
- Los URI asociados a los recursos pueden cambiar si modificamos el recurso (nombre, ubicación, características, etc)

Métodos HTTP

HTTP	REST	Descripción
GET	RETRIEVE	Sin identificador: Recuperar el estado completo de un recurso (HEAD + BODY) Con identificador: Recuperar el estado individual de un recurso (HEAD + BODY)
HEAD		Recuperar el estado de un recurso (HEAD)
POST	CREATE or REPLACE	Crea o modifica un recurso (sin identificador)
PUT	CREATE or REPLACE	Crea o modifica un recurso (con identificador)
DELETE	DELETE	Sin identificador: Elimina todo el recurso Con identificador: Elimina un elemento concreto del recurso
CONNECT		Comprueba el acceso al host
TRACE		Solicita al servidor que introduzca en la respuesta todos los datos que reciba en el mensaje de petición
OPTIONS		Devuelve los métodos HTTP que el servidor soporta para un URL específico
PATCH	REPLACE	HTTP 1.1 Reemplaza parcialmente un elemento del recurso

Tipos MIME

- Otro aspecto muy importante es la posibilidad de negociar distintos formatos (representaciones) a usar en la transferencia del estado entre servidor y cliente (y viceversa).
- La representación de los recursos es el formato de lo que se envía un lado a otro entre clientes y servidores.
- Con HTTP podemos transferir múltiples tipos de información.
- Los datos se transmiten a través de TCP/IP, el navegador sabe cómo interpretar las secuencias binarias (Content-Type) por el protocolo HTTP
- La representación de un recurso depende del tipo de llamada que se ha generado (Texto, HTML, PDF, etc).
- En HTTP cada uno de estos formatos dispone de su propio tipos MIME, en el formato <tipo>/<subtipo>.
  - application/json application/xml text/html text/plain image/jpeg

Tipos MIME

- Para negociar el formato entre el cliente y el servidor se utilizan las cabeceras:
  - Petición
    - En la cabecera ACCEPT se envía una lista de tipos MIME que el cliente entiende.
    - En caso de enviar contenido en el cuerpo, la cabecera CONTENT-TYPE indica en que formato MIME está codificado.
  - Respuesta
    - El servidor selecciona el tipo que más le interese de entre todos los especificados en la cabecera ACCEPT, y devuelve la respuesta indicando con la cabecera CONTENT-TYPE el formato del cuerpo.
- La lista de tipos MIME se especifica en la cabecera (ACCEPT) mediante lo que se llama una lista separada por comas de tipos (media range). También pueden aparecer expresiones de rango, por ejemplo
  - */* indica cualquier tipo MIME
  - image / * indica cualquier formato de imagen
- Si el servidor no entiende ninguno de los tipos MIME propuestos (ACCEPT) devuelve un mensaje con código 406 (incapaz de aceptar petición).

Códigos HTTP (status)

status	statusText	Descripción
100	Continue	Una parte de la petición (normalmente la primera) se ha recibido sin problemas y se puede enviar el resto de la petición
101	Switching protocols	El servidor va a cambiar el protocolo con el que se envía la información de la respuesta. En la cabecera Upgrade indica el nuevo protocolo
200	OK	La petición se ha recibido correctamente y se está enviando la respuesta. Este código es con mucha diferencia el que mas devuelven los servidores
201	Created	Se ha creado un nuevo recurso (por ejemplo una página web o un archivo) como parte de la respuesta
202	Accepted	La petición se ha recibido correctamente y se va a responder, pero no de forma inmediata
203	Non-Authoritative Information	La respuesta que se envía la ha generado un servidor externo. A efectos prácticos, es muy parecido al código 200
204	No Content	La petición se ha recibido de forma correcta pero no es necesaria una respuesta
205	Reset Content	El servidor solicita al navegador que inicialice el documento desde el que se realizó la petición, como por ejemplo un formulario
206	Partial Content	La respuesta contiene sólo la parte concreta del documento que se ha solicitado en la petición

Códigos de redirección

status	statusText	Descripción
300	Multiple Choices	El contenido original ha cambiado de sitio y se devuelve una lista con varias direcciones alternativas en las que se puede encontrar el contenido
301	Moved Permanently	El contenido original ha cambiado de sitio y el servidor devuelve la nueva URL del contenido. La próxima vez que solicite el contenido, el navegador utiliza la nueva URL
302	Found	El contenido original ha cambiado de sitio de forma temporal. El servidor devuelve la nueva URL, pero el navegador debe seguir utilizando la URL original en las próximas peticiones
303	See Other	El contenido solicitado se puede obtener en la URL alternativa devuelta por el servidor. Este código no implica que el contenido original ha cambiado de sitio
304	Not Modified	Normalmente, el navegador guarda en su caché los contenidos accedidos frecuentemente. Cuando el navegador solicita esos contenidos, incluye la condición de que no hayan cambiado desde la última vez que los recibió. Si el contenido no ha cambiado, el servidor devuelve este código para indicar que la respuesta sería la misma que la última vez
305	Use Proxy	El recurso solicitado sólo se puede obtener a través de un proxy, cuyos datos se incluyen en la respuesta
307	Temporary Redirect	Se trata de un código muy similar al 302, ya que indica que el recurso solicitado se encuentra de forma temporal en otra URL

Códigos de error del navegador

status	statusText	Descripción
400	Bad Request	El servidor no entiende la petición porque no ha sido creada de forma correcta
401	Unauthorized	El recurso solicitado requiere autorización previa
402	Payment Required	Código reservado para su uso futuro
403	Forbidden	No se puede acceder al recurso solicitado por falta de permisos o porque el usuario y contraseña indicados no son correctos
404	Not Found	El recurso solicitado no se encuentra en la URL indicada. Se trata de uno de los códigos más utilizados y responsable de los típicos errores de *Página no encontrada*
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta

Códigos de error del navegador

status	statusText	Descripción
409	Conflict	El navegador no puede procesar la petición, ya que implica realizar una operación no permitida (como por ejemplo crear, modificar o borrar un archivo)
410	Gone	Similar al código 404. Indica que el recurso solicitado ha cambiado para siempre su localización, pero no se proporciona su nueva URL
411	Length Required	El servidor no procesa la petición porque no se ha indicado de forma explícita el tamaño del contenido de la petición
412	Precondition Failed	No se cumple una de las condiciones bajo las que se realizó la petición
413	Request Entity Too Large	La petición incluye más datos de los que el servidor es capaz de procesar. Normalmente este error se produce cuando se adjunta en la petición un archivo con un tamaño demasiado grande
414	Request-URI Too Long	La URL de la petición es demasiado grande, como cuando se incluyen más de 512 bytes en una petición realizada con el método GET
415	Unsupported Media Type	Al menos una parte de la petición incluye un formato que el servidor no es capaz procesar
416	Requested Range Not Suitable	El trozo de documento solicitado no está disponible, como por ejemplo cuando se solicitan bytes que están por encima del tamaño total del contenido
417	Expectation Failed	El servidor no puede procesar la petición porque al menos uno de los valores incluidos en la cabecera Expect no se pueden cumplir

Códigos de error del servidor

status	statusText	Descripción
500	Internal Server Error	Se ha producido algún error en el servidor que impide procesar la petición
501	Not Implemented	Procesar la respuesta requiere ciertas características no soportadas por el servidor
502	Bad Gateway	El servidor está actuando de proxy entre el navegador y un servidor externo del que ha obtenido una respuesta no válida
503	Service Unavailable	El servidor está sobrecargado de peticiones y no puede procesar la petición realizada
504	Gateway Timeout	El servidor está actuando de proxy entre el navegador y un servidor externo que ha tardado demasiado tiempo en responder
505	HTTP Version Not Supported	El servidor no es capaz de procesar la versión HTTP utilizada en la petición. La respuesta indica las versiones de HTTP que soporta el servidor

Módulos

- Las interfaces HTTP en Node.js están diseñadas para admitir muchas características del protocolo que tradicionalmente han sido difíciles de usar. En particular, mensajes grandes, posiblemente codificados en fragmentos. La interfaz tiene cuidado de no almacenar nunca solicitudes o respuestas completas; el usuario puede transmitir datos.
- Node.js dispone de varios módulos base y utilidades para implementar servidores y aplicaciones web: HTTP, HTTP/2, HTTPS, TLS/SSL, DNS, URL, Query Strings

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hola mundo.\n');
});

server.listen(3000, '127.0.0.1');
```

https://expressjs.com

EXPRESS

Introducción

- Express es una infraestructura de aplicaciones web Node.js mínima y flexible que proporciona un conjunto sólido de características para las aplicaciones web y móviles.
- Sencillo y flexible, simplifica enormemente el desarrollo web con Node, envolviendo los módulos base (HTTP, URL, DNS, …) hace innecesario su uso.
- Se ha convertido en el framework de referencia en el que se basan los framework mas populares.
- Express nos va ayudar con rutas, parámetros, templates, formularios, subida/bajada de ficheros, cookies, sesiones.
- Express.js está basado en Connect, que a su vez es un framework basado en http para Node.js. Podemos decir que Connect tiene todas las opciones del módulo http que viene por defecto con Node y le suma funcionalidades. A su vez, Express hace lo mismo con Connect, con lo que tenemos un framwork ligero, rápido y muy útil

Instalación

- Para agregar a una aplicación existente:
  - npm install express --save
- Se puede utilizar la herramienta generadora de aplicaciones, express-generator, para crear rápidamente un esqueleto de aplicación.
  - npx express-generator --view=pug --git myapp
- Si se usa frecuentemente se puede instalar globalmente:
  - npm install -g express-generator
  - express --view=pug --git myapp
- Mediante opciones se puede establecer la ingeniería de plantillas, CSS, git, … utilizadas al generar.
- El generador solo crea la infraestructura, es necesario instalar manualmente las dependencias.
  - cd myapp & npm install
- Para ejecutar con la traza de depuración activada:
  - set DEBUG=myapp:* & npm start

Estructura

- app.js
- package.json
- bin
  - www
- Public
  - Images
  - Javascripts
  - stylesheets
    - style.css
- routes
  - index.js
  - users.js
- views
  - error.pug
  - index.pug
  - layout.pug

Anatomía de una aplicación

- Una aplicación Express:
  - debe cargar el módulo,
  - crear el objeto aplicación,
  - configurar los middleware en la aplicación,
  - agregar el enrutamiento,
  - iniciar el servidor y escucha las conexiones del puerto
- La aplicación principal

```
const express = require('express')
const app = express()
const port = 3000

// configuración + rutas

app.listen(port, () => console.log(`App listening on port ${port}!`))
```


Enrutamiento

- El enrutamiento se refiere a determinar cómo una aplicación responde a una solicitud del cliente a un punto final particular, que es una URI (o ruta) y un método de solicitud HTTP específico (GET, POST, etc.). Cada ruta puede tener una o más funciones de controlador, que se ejecutan cuando la ruta coincide.
- La definición de ruta toma la siguiente estructura:
  - app.METHOD(PATH, HANDLER)
- Dónde:
  - METHOD es un método de solicitud HTTP (GET, POST, …), en minúsculas.
  - PATH es una ruta en el servidor.
  - HANDLER es la función ejecutada cuando la ruta coincide.

```
app.get('/', (req, res) => res.send('Hello World! '))
```

- Para servir archivos estáticos como imágenes, archivos CSS y archivos JavaScript:

```
app.use(express.static(path.join(__dirname, 'public')))
app.use('/static', express.static(path.join(__dirname, 'public')))
```

Rutas

- Las trayectorias de ruta, en combinación con un método de solicitud, definen los puntos finales en los que se pueden realizar solicitudes. Las trayectorias de ruta pueden ser cadenas, patrones de cadena o expresiones regulares.
- Los caracteres ?, +, *, y () son un subconjunto de sus homólogos de expresiones regulares. El guion (-) y el punto (.) se interpretan literalmente en las rutas basadas en cadenas.
- Para utilizar el carácter de dólar en una cadena de ruta hay que expresarlo como ([\$]): /data/([\$])book → /data/$book.
- Los parámetros de ruta se denominan segmentos de URL que se utilizan para capturar los valores especificados en su posición en la URL. Los valores capturados se rellenan en el objeto req.params, con el nombre del parámetro en la ruta como propiedad.

```
app.get('/users/:userId/books/:bookId', function (req, res) {
```


Rutas

- Se pueden crear controladores de ruta encadenables para una trayectoria de ruta utilizando app.route(). Debido a que la ruta se especifica en una única ubicación, la creación de rutas modulares es útil, por la reducción de redundancia y errores tipográficos.

```
app.route('/book')
  .get(function (req, res) {
    res.send('Get a random book')
  })
  .post(function (req, res) {
    res.send('Add a book')
  })
  .put(function (req, res) {
    res.send('Update the book')
  })
```

Rutas

- Para modularizar la aplicación y disminuir la complejidad de la aplicación, se pueden crear módulos específico para cada ruta, a menudo se les conoce como una "mini aplicación".

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

- Para luego cargar el módulo enrutador en la aplicación:

```
var indexRouter = require('./routes/index');
// …
app.use('/', indexRouter);
```


Manejadores de ruta

- Cada ruta debe estar asociada a un manejador que genera la respuesta, para ello recibe dos parámetros: petición, respuesta.

```
(req, res) => { … }
```

- Se puede proporcionar múltiples manejadores que se comporten como middleware para manejar una solicitud. Los manejadores reciben un tercer parámetro con la función next para invocar el siguiente manejador, pasar el control a rutas posteriores si no hay razón para continuar con la ruta actual o, si no se invoca, omitir el resto de los manejadores de la ruta.

```
function (req, res, next) {
  // …
  next()
}
```

- Los manejadores de ruta pueden tener la forma de una función, una matriz de funciones o combinaciones de ambas.

Petición

- El parámetro req es el objeto que representa la solicitud HTTP y permite acceder a la información de la petición, tiene propiedades para la cadena de consulta, parámetros, cuerpo, encabezados, contexto, etc.
- Para acceder a los parámetros de la ruta:

```
app.get('/users/:userId/books/:bookId', function (req, res) {
  let book = find(req.params.userId, req.params.bookId);
```

- Para obtener un parámetro de cadena de consulta en la ruta:

```
// GET: /users/1?page=1&size=20
var lst = filter(req.query.page, req.query.size);
```

- Para acceder a las cabeceras:

```
req.headers.referer → req.get('referer')
req.get('content-type') === 'application/json' → req.is('application/json')
req.accepts('application/json')
```

- Cuando se utiliza el middleware cookie-parser, para acceder a las cookies enviadas por la solicitud:

```
var sid = req.cookies.sid;
```


Petición

- req.body contiene pares de datos clave-valor enviados en el cuerpo de la solicitud. De forma predeterminada está undefined y se completa cuando se utiliza un middleware de análisis del cuerpo como express.json() o express.urlencoded().

```
app.use(express.json()) // for parsing application/json
app.use(express.urlencoded({ extended: true })) // for parsing application/x-www-form-urlencoded

app.post('/profile', function (req, res, next) {
  console.log(req.body)
  res.json(req.body)
})
```

- Para acceder a la información de contexto se dispone de las propiedades: app, baseUrl, originalUrl, ip, ips, protocol, hostname, method, path, xhr, …

```
app.use('/admin', function (req, res, next) { // GET 'http://www.example.com/admin/new'
  console.dir(req.originalUrl) // '/admin/new'
  console.dir(req.baseUrl) // '/admin'
  console.dir(req.path) // '/new'
  next()
})
```

Respuesta

- El parámetro res es el objeto que representa la respuesta HTTP que envía una aplicación Express cuando recibe una solicitud HTTP.
- Los métodos del objeto de respuesta permiten preparar y enviar una respuesta al cliente y finalizar el ciclo de solicitud-respuesta.
  - res.send(): Enviar una respuesta de varios tipos.
  - res.json(): Enviar una respuesta JSON.
  - res.sendFile(): Enviar un archivo como una secuencia de octetos.
  - res.render(): Renderizar una plantilla de vista.
  - res.download(): Solicitar un archivo para descargar.
  - res.status(): Establezca el código de estado de respuesta.
  - res.sendStatus(): Establezca el código de estado de respuesta y envíe su representación de cadena como el cuerpo de respuesta.
  - res.redirect(): Redirigir una solicitud.
  - res.end(): Terminar el proceso de respuesta.
- Si ninguno de estos métodos se llama desde un controlador de ruta, la solicitud del cliente quedará pendiente.

Respuesta

- Para establecer las cabeceras:

```
res.append('cache-control', 'max-age=600')
res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
res.type('application/json')
```

- Para enviar las cookies:

```
res.status(201)
  .cookie('access_token', 'Bearer ' + token, {
    expires: new Date(Date.now() + 8 * 3600000)
  }).redirect(301, '/admin')
```

Respuesta

- res.format permite realizar la negociación de contenido en el encabezado HTTP Accept de la solicitud, cuando está presente. Utiliza req.accepts () para seleccionar un controlador para la solicitud, en función de los tipos aceptables priorizados.
- Si no se especifica el encabezado, se invoca la primera devolución de llamada.
- Cuando no se encuentra ninguna coincidencia, el servidor responde con 406 "No aceptable" o invoca la devolución de llamada default.
- Cuando se selecciona una devolución de llamada también se establece el encabezado Content-Type de la respuesta.

```
res.format({
  'text/plain': function () { res.send('hey') },
  'text/html': function () { res.send('<p>hey</p>')  },
  'application/json': function () { res.json({ message: 'hey' }) },
  'default': function () { res.status(406).send('Not Acceptable') }
})
```


Middleware

- Express es un marco web de enrutamiento y middleware que tiene una funcionalidad mínima propia: una aplicación Express es esencialmente una serie de llamadas a funciones de middleware.
- Las funciones de middleware son funciones que tienen acceso al objeto de solicitud (req), el objeto de respuesta (res) y la función next en el ciclo de solicitud-respuesta de la aplicación. La función next es la función que permite controlar el flujo de operaciones sucesivas.
- Las funciones de middleware pueden realizar las siguientes tareas:
  - Ejecutar cualquier código.
  - Realizar cambios en la solicitud y los objetos de respuesta.
  - Finalizar el ciclo de solicitud-respuesta.
  - Llamar al siguiente middleware en la pila.
- Si la función de middleware actual no finaliza el ciclo de solicitud-respuesta, debe llamar a next() para pasar el control a la siguiente función de middleware. De lo contrario, la solicitud quedará pendiente.
- Los Middleware son módulos “plug and play” que proveen cierta funcionalidad y se pueden apilar arbitrariamente en cualquier orden.

Middleware de nivel de aplicación

- Para enlazar el middleware de nivel de aplicación a una instancia del objeto de la aplicación se utiliza las funciones app.use() y app.METHOD(), donde METHOD está el método HTTP de la solicitud que maneja la función de middleware (como GET, PUT o POST) en minúsculas.
- Para que la función se ejecute cada vez que la aplicación recibe una solicitud:

```
app.use(function (req, res, next) {
  res.locals.time = Date.now()
  next()
})
```

- Para que la función se ejecute cada vez que la aplicación recibe una solicitud a una ruta:

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```


Middleware de nivel de enrutador

- El middleware de nivel de enrutador funciona de la misma manera que el middleware de nivel de aplicación, excepto que está vinculado a una instancia de express.Router().
- Se carga el middleware a nivel de enrutador utilizando las funciones router.use() y router.METHOD().

```
const express = require('express')
var router = express.Router()

router.use(function (req, res, next) {
   if (req.get('Authorization')) {
      res.locals.token = req.get('Authorization')
      next();
   } else
      res.status(401).end('No autenticado.');
})
router.use('/', function (req, res, next) {
   res.send('Modo administrador: '+ res.locals.token)
   next()
})

module.exports = router;
```

- Para omitir el resto de las funciones de middleware del enrutador, hay que llamar a next('router') para pasar el control fuera de la instancia del enrutador.

Middleware de manejo de errores

- El middleware para el manejo de errores siempre toma cuatro argumentos. Se debe proporcionar cuatro argumentos para identificarlo como una función de manejo de errores. Incluso si no se necesita usar el objeto next, se debe especificarlo para mantener la firma porque de lo contrario se interpretará como middleware normal y no se podrán manejar los errores.
- Las funciones de middleware para el manejo de errores se definen de la misma manera que otras funciones de middleware, excepto por la firma con cuatro parámetros en lugar de tres: (err, req, res, next):

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


Middleware incorporado

- A partir de la versión 4.x, Express ya no depende de Connect. Las funciones de middleware que se incluían anteriormente con Express ahora están en módulos separados.
- Express tiene las siguientes funciones de middleware integradas:
  - express.static sirve activos estáticos como archivos HTML, imágenes, etc.
  - express.json analiza las solicitudes entrantes con cargas JSON. NOTA: Disponible con Express 4.16.0+
  - express.urlencoded analiza las solicitudes entrantes con cargas útiles codificadas en URL. NOTA: Disponible con Express 4.16.0+

Middleware de terceros

- Se puede utilizar middleware de terceros para agregar funcionalidad a las aplicaciones Express.
- Para utilízalos es necesario:
  - Instalar con npm el módulo Node.js para la funcionalidad requerida,
  - Cargar con require el modulo
  - Añadirlo a la aplicación en el nivel de la aplicación o en el nivel del enrutador.
- Por ejemplo, para la instalación y carga de la función de middleware cookie-parser de análisis de cookies.

```
$ npm install cookie-parser

var express = require('express')
var app = express()
var cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```


Middleware de terceros

Módulo	Descripción
body-parser	Analizar el cuerpo de la solicitud HTTP.
compression	Comprime las respuestas HTTP.
connect-rid	Generar ID de solicitud única.
cookie-parser	Analizar el encabezado de la cookie y rellena req.cookies.
cookie-session	Establecer sesiones basadas en cookies.
cors	Habilitar el uso compartido de recursos de origen cruzado (CORS) con varias opciones.
csurf	Proteger de las vulnerabilidades CSRF.
errorhandler	Desarrollo de manejo de errores / depuración.
method-override	Anular los métodos HTTP usando el encabezado.

Middleware de terceros

Módulo	Descripción
morgan	Registrador de solicitudes HTTP.
multer	Manejar datos de formularios de varias partes.
response-time	Registrar el tiempo de respuesta HTTP.
serve-favicon	Servir un favicon.
serve-index	Servir el listado de directorio para una ruta determinada.
serve-static	Servir archivos estáticos.
session	Establecer sesiones basadas en servidor (solo desarrollo).
timeout	Establecer un período de tiempo de espera para el procesamiento de solicitudes HTTP.
vhost	Crea dominios virtuales.

Manejo de errores

- La gestión de errores se refiere a cómo Express detecta y procesa los errores que se producen de forma sincrónica y asincrónica. Express viene con un controlador de errores predeterminado, por lo que no se necesita escribir uno propio para comenzar.
- Es importante asegurarse de que Express detecta todos los errores que se producen al ejecutar controladores de ruta y middleware.
- Los errores que ocurren en el código sincrónico dentro de los controladores de ruta y middleware no requieren trabajo adicional. Si el código síncrono arroja un error, Express lo detectará y procesará.

```
app.get('/', function (req, res) {
  throw new Error('BROKEN') // Express will catch this on its own.
})
```

- Para los errores devueltos por funciones asincrónicas invocadas por controladores de ruta y middleware, deben interceptarse y pasarlos a la función next(), donde Express los detectará y procesará:

```
app.get('/', function (req, res, next) {
  fs.readFile('/file-does-not-exist', function (err, data) {
    if (err) { next(err) } else { res.json(data) }
  })
})
```

- Si se pasa algo a la función next() (excepto la cadena 'route'), Express considera que la solicitud actual es un error y omitirá cualquier ruta de enrutamiento y funciones de middleware restantes sin manejo de errores.

Manejo de errores

- Express viene con un controlador de errores incorporado que se encarga de cualquier error que pueda encontrar en la aplicación. Esta función de middleware de gestión de errores predeterminada se agrega al final de la pila de funciones de middleware.
- Si hay un error next() y no se maneja en un controlador de errores personalizado, será manejado por el controlador de errores incorporado: El error se escribirá en el cliente con el seguimiento de la pila (el seguimiento de la pila no está incluido en el entorno de producción).
- Si se llama next() con un error después de haber comenzado a escribir la respuesta (por ejemplo, si encuentra un error mientras transmite la respuesta al cliente), el controlador de error predeterminado Express cierra la conexión y falla la solicitud.
- Cuando se agrega un controlador de errores personalizado, se debe delegar al controlador de errores Express predeterminado cuando los encabezados ya se hayan enviado al cliente:

```
function errorHandler (err, req, res, next) {
  if (res.headersSent) {
    return next(err)
  }
  res.status(500)
  res.render('error', { error: err })
}
```

- Hay que tener en cuenta que el controlador de errores predeterminado puede activarse si se llama next() con un error en el código más de una vez, incluso si el middleware de manejo de errores personalizado está en su lugar.

Manejo de errores

- Se pueden definir las funciones de middleware de gestión de errores de la misma manera como otras funciones de middleware, excepto las funciones de control de errores tienen cuatro argumentos en lugar de tres: (err, req, res, next).

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

- Las respuestas desde una función de middleware pueden estar en cualquier formato, como una página de error HTML, un mensaje simple o una cadena JSON.
- Para fines de organización (y marco de nivel superior), se pueden definir varias funciones de middleware para el manejo de errores, tal como lo haría con las funciones de middleware normales.

```
app.use(logErrors)
app.use(clientErrorHandler)
app.use(errorHandler)
```

- Hay que tener en cuenta que cuando no se llama next() en una función de manejo de errores, se es responsable de escribir (y finalizar) la respuesta. De lo contrario, esas solicitudes se "colgarán" y no serán elegibles para la recolección de basura.

TEMPLATES

Introducción

- Un motor de plantillas permite usar archivos de plantillas estáticas en la aplicación. En tiempo de ejecución, el motor de plantillas reemplaza las variables en un archivo de plantilla con valores reales y transforma la plantilla en un archivo HTML que envía al cliente. Este enfoque facilita el diseño de una página HTML.
- Algunos motores de plantillas populares que funcionan con Express son Jade, Swig, Pug , Moustache y EJS . El generador de aplicaciones Express usa Jade como predeterminado, pero también es compatible con varios otros.
- Jade ha cambiado de nombre a Pug. Se puede continuar usando Jade en la aplicación, y funcionará bien. Sin embargo, si se desea las últimas actualizaciones para el motor de plantillas, se debe reemplazar Jade con Pug en su aplicación.
- Los motores de plantillas se pueden utilizar de forma independiente o integrados con Express.
- Para descargar e instalar el motor de plantillas:
  - npm install pug --save

Instalación y configuración

- Para utilizar las plantillas hay que:
  - Instalar el paquete de npm del motor de plantilla correspondiente.
  - Es conveniente crear un directorio para organizar las plantillas.
  - Establecer el directorio donde se encuentran los archivos de plantilla.
  - Establecer la ingeniería del motor de plantillas para usar.

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

- Después de configurar el motor de vista, no se tiene que especificar el motor o cargar el módulo de motor de plantilla en la aplicación, Express carga el módulo internamente.

Uso de plantillas

- Crear un archivo de plantilla nombrado demo.pug (la extensión del fichero depende del motor de plantillas) en el directorio views:

```
html
  head
    title= title
  body
    h1= message
```

- Para representar la plantilla:

```
app.get('/demos', (req, res) => {
  res.render('demo', { title: 'Curso', message: 'Hola mundo!' })
});
```

- Si la propiedad view engine no está establecida se debe especificar la extensión del archivo, si no, se puede omitirlo.

Pug (Jade)

- Pug (anteriormente conocido como Jade) es un sistema de plantillas inspirado en JavaScript y en Haml para utilizarse en Node.js.
- Pug en un lenguaje de plantillas, un framework de NodeJS y una forma rápida de generar contenido. Básicamente define una estructura semántica, ordenada y jerárquica, usando solamente el nombre de las etiquetas HTML, de esta forma solamente nos dedicamos a generar contenido y no en aprender un lenguaje nuevo, dado que es una abstracción de HTML plasmada en un formato más sencillo, de aquí deriva el "nuevo" lenguaje llamado Pug.
- Pros
  - No hay etiquetas de cierre
  - Basado en el sangrado
  - Herencia de diseño
  - Soporte de macros
  - Includes de la vieja escuela
  - Soporte integrado para Markdown, CoffeeScript y otros.
  - Implementaciones disponibles en JavaScript, php , scala , ruby , python y java.
- Contras
  - No a todos les gusta el sangrado con espacios en blanco significativos
  - La incorporación de JavaScript puede ser engorrosa para más de una línea
  - Requiere un pequeño runtime para usar plantillas pre compiladas en el cliente
  - No es adecuado para salidas que no sean HTML
  - Sin soporte de streaming
  - Curva de aprendizaje algo alta

Sintaxis

- Se basa en escribir una etiqueta en cada línea y sangrar su contenido, la etiqueta se cierra automáticamente cuando se reduce el sangrado.
- El código se reduce considerablemente debido a que se escribe una sola vez la etiqueta deseada, evitando así abrir y cerrar etiquetas HTML, los caracteres '<' y '>' no son necesarios.
- Los atributos de la etiqueta se expresan entre paréntesis con pares atributo='valor' separados por espacios o comas.

```
h1(style={color: 'white', background: 'darkred'})= message
  form(action="frm", method="post", class=['foo', 'bar', 'baz'])
    label(for="usr") Usuario
    input#usr(type="text", name="usr")
    input(type='checkbox' checked)
    | Recordar
    button(type="submit") Iniciar
```

- Los IDs pueden definirse con #idname y las clases CSS con .classname (se pueden encadenar varias consecutivas).
- Todo después de la etiqueta y un espacio serán los contenidos de texto de esa etiqueta. Si se inicia una línea con un carácter de barra (|) continua el contenido anterior.

Valores

- Al invocar el método render se puede inyectar un objeto de datos (llamado " locals") que contiene la información a representar en la plantilla.
- La plantilla puede declarar sus propias variables, que tendrán como ámbito de la propia plantilla.
- El código sin búfer comienza con -. No agrega directamente nada a la salida.
- Las variables se declaran con var y como código sin buffer:
  - var url = 'https://example.com/'
  - var attributes = { tittle: 'foo' };
  - attributes.class = 'baz';
  - var list = ["Uno", "Dos", "Tres", "Cuatro", "Cinco", "Seis"]

Interpolación

- Las plantillas permiten introducir expresiones que serán evaluadas y sustituidas por su valor al representar la plantilla.
- Siguen el patrón básico para evaluar una plantilla local, pero el código intermedio #{ y } se evalúa, escapa y el resultado se almacena en la salida de la plantilla que se representa.

  h1 Hola #{nombre}
- Para asignar el contenido completo:

  h1= message
- El resultado se escapa por defecto, para evitarlo se utiliza !:

  p El titulo es !{'' + title + ''}

  h1!= message
- La interpolación no solo funciona en valores de JavaScript, también permite expresiones de plantilla para generar etiquetas, envolviéndola en #[y]:

  p El titulo es #[b.titulo #{title}]

Condicionales

- Condicional simple:

```
if authorised
  p Autorizado
else
  p Anonimo
```

- Condicional simple negada:

```
unless authorised
  p Anonimo
```

- Condicional múltiple:

```
case friends
  when 0
    p you have no friends
  when 1
    p you have a friend
  default
    p you have #{friends} friends
```


Bucles

```
ol
  - for (var x = 0; x < 3; x++)
    li ítem

ul
  each val in [1, 2, 3, 4, 5]
    li= val
ul
  each val, index in {1:'one',2:'two',3:'three'}
    li= index + ': ' + val
ul
  each val in values
    li= val
  else
    li There are no values
- var n = 0;

ul
  while n < 4
    li= n++
```

Mixins

- Los mixins permiten crear bloques reutilizables. Pueden tomar argumentos y capturar el contenido como un bloque:

```
mixin article(title)
  .article
    h1= title
    if block
      block
    else
      p No content provided

+article('Hello world')

+article('My article')
  p This is my
  p Amazing article
```


Includes

- include le permite insertar el contenido de un archivo en otro.

```
doctype html
html
  include includes/head
  style
    include style.css
  body
    h1 My Site
    p Welcome to my super lame site.
    include includes/foot
```

Template Inheritance

- La herencia de plantillas funciona mediante las palabras clave block y extends.
- En una plantilla, un block es simplemente un "bloque" que una plantilla secundaria puede reemplazar. Este proceso es recursivo.
- Los bloques pueden proporcionar contenido predeterminado, si corresponde.

```
html
  head
    title My Site - #{title}
    block scripts
      script(src='/jquery.js')
  body
    block content
    block foot
      #footer
        p some default footer content
```


Template Inheritance

- Las plantillas extendidas utilizan la directiva extends referenciar la plantilla principal y definen uno o más bloques para anular el contenido del bloque principal.

```
extends layout

block scripts
  script(src='/jquery.js')
  script(src='/pets.js')

block content
  h1= title
  - var pets = ['cat', 'dog']
  each petName in pets
    include pet
```

- Las plantillas extendidas pueden definir a su vez bloques para sus sub plantillas.

Template Inheritance

- Los bloques de las plantillas extendidas se pueden definir como:
  - replace(por defecto): sustituye el bloque heredado por el de la plantilla extendida.
  - prepend: añade el valor por defecto de bloque heredado al de la plantilla extendida.

```
block prepend scripts
  script(src='/game.js')
// <script src="/game.js"></script><script src="/jquery.js"></script>
```

  - append: añade el bloque de la plantilla extendida al valor por defecto de bloque heredado.

```
block append scripts
  script(src='/game.js')
// <script src="/jquery.js"></script><script src="/game.js"></script>
```

- Cuando se usa block append o block prepend, la palabra " block" es opcional.

PERSISTENCIA

GenerateData

- GenerateData es una herramienta para la generación automatizada de juegos de datos.
- Ofrece ya una serie de datos precargados en BBDD y un conjunto de tipos de datos bastante amplio, así como la posibilidad de generar tipos genéricos.
- Podemos elegir en que formato se desea la salida de entre los siguientes:
  - CSV
  - Excel
  - HTML
  - JSON
  - LDIF
  - Lenguajes de programación (JavaScript, Perl, PHP, Ruby, C#)
  - SQL (MySQL, Postgres, SQLite, Oracle, SQL Server)
  - XML
- Online: http://www.generatedata.com/?lang=es
- Instalación (PHP): http://benkeen.github.io/generatedata/

Mockaroo

- https://www.mockaroo.com/
- Mockaroo permite descargar rápida y fácilmente grandes cantidades de datos realistas de prueba generados aleatoriamente en función de sus propias especificaciones que luego puede cargar directamente en su entorno de prueba utilizando formatos CSV, JSON, XML, Excel, SQL, … No se requiere programación y es gratuita (para generar datos de 1.000 en 1.000).
- Los datos realistas son variados y contendrán caracteres que pueden no funcionar bien con nuestro código, como apóstrofes o caracteres unicode de otros idiomas. Las pruebas con datos realistas harán que la aplicación sea más robusta porque detectará errores en escenarios de producción.
- El proceso es sencillo ya que solo hay que ir añadiendo nombres de campos y escoger su tipo. Por defecto nos ofrece mas de 140 tipos de datos diferentes que van desde nombre y apellidos (pudiendo escoger estilo, género, etc...) hasta ISBNs, ubicaciones geográficas o datos encriptados simulados.
- Además es posible hacer que los datos sigan una distribución Normal o de Poisson, secuencias, que cumplan una expresión regular o incluso que fuercen cadenas complicadas con caracteres extraños y cosas así. Tenemos la posibilidad de crear fórmulas propias para generarlos, teniendo en cuenta otros campos, condicionales, etc... Es altamente flexible.

Sistema de archivos

- El módulo fs proporciona una API para interactuar con el sistema de archivos de una manera muy similar a las funciones estándar POSIX.

```
const fs = require('fs');
```

- Todas las operaciones del sistema de archivos tienen formas sincrónicas (con sufijo Sync) y asincrónicas.
- La forma asincrónica siempre tienen una función callback de finalización como último argumento. Los argumentos pasados a la función callback dependen del método, pero el primer argumento siempre está reservado para la excepción. Si la operación se completó con éxito, el primer argumento será null o undefined.

```
const fs = require('fs');

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

- No está garantizado el orden en que se ejecutan los callback cuando se utilizan métodos asincrónicos sucesivos, salvo que las operaciones sucesivas se vayan anidando en los callback.

Sistema de archivos

- Las versiones síncronas bloquearán todo el proceso hasta que se completen, deteniendo todas las conexiones. Por ello no requieren funciones callback y se garantiza el orden de ejecución. Dado que los procesos I/O son lentos por naturaleza y Node se ejecuta en un único hilo, no se recomienda el uso de las versiones síncronas que degradan seriamente el rendimiento.
- Las excepciones que ocurren usando operaciones sincrónicas se generan de inmediato y se pueden manejar usando try…catch, o se puede permitir que burbujeen.

```
const fs = require('fs');

try {
  fs.unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}
```

Rutas de archivo

- La mayoría de las operaciones fs aceptan rutas de archivo que pueden especificarse en forma de una cadena, un Buffer o un objeto URL.
- Las rutas en forma de cadena se interpretan como secuencias de caracteres UTF-8 que identifican el nombre de archivo absoluto o relativo. Las rutas relativas se resolverán en relación con el directorio de trabajo actual según lo especificado por process.cwd().
  - fs.open('/open/some/file.txt', 'r', (err, fd) => {
- Las rutas especificadas mediante a Buffer son útiles principalmente en ciertos sistemas operativos POSIX que tratan las rutas de archivos como secuencias de bytes opacas. En tales sistemas, es posible que una sola ruta de archivo contenga subsecuencias que usan codificaciones de caracteres múltiples. Pueden ser relativas o absolutas:
  - fs.open(Buffer.from('/open/some/file.txt'), 'r', (err, fd) => {
- En Windows, Node.js sigue el concepto de directorio de trabajo por unidad. Este comportamiento se puede observar cuando se utiliza una ruta de unidad sin una barra invertida: 'c:\\' puede devolver un resultado diferente que 'c:'.

Soporte de rutas URL

- Para la mayoría de las funciones del módulo fs, el argumento path o filename se puede pasar como un objeto URL WHATWG. Solo se admiten objetos URL que utilizan el protocolo file://.

```
const fs = require('fs');
const fileUrl = new URL('file:///tmp/hello');
fs.readFileSync(fileUrl);
```

- Las URL son siempre rutas absolutas.
- El uso de objetos URL WHATWG podría introducir comportamientos específicos de la plataforma.
- En Windows, las URL file: con un nombre de host se convierten en rutas UNC, mientras que las URL file: con letras de unidad se convierten en rutas absolutas locales. Las URL file: sin nombre de host ni letra de unidad generarán una excepción.
  - file://hostname/p/a/t/h/file → \\hostname\p\a\t\h\file
  - file:///C:/tmp/hello → C:\tmp\hello

Descriptores de archivo

- En los sistemas POSIX, para cada proceso, el núcleo mantiene una tabla de archivos y recursos actualmente abiertos. A cada archivo abierto se le asigna un identificador numérico simple llamado descriptor de archivo. A nivel del sistema, todas las operaciones del sistema de archivos usan estos descriptores de archivos para identificar y rastrear cada archivo específico. Los sistemas Windows utilizan un mecanismo diferente pero conceptualmente similar para el seguimiento de los recursos. Para simplificar las cosas, Node.js abstrae las diferencias específicas entre los sistemas operativos y asigna a todos los archivos abiertos un descriptor numérico.
- El método fs.open() se utiliza para asignar un nuevo descriptor de archivo. Una vez asignado, el descriptor de archivo puede usarse para leer datos, escribir datos o solicitar información sobre el archivo.

```
fs.open('/open/some/file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.fstat(fd, (err, stat) => {
    if (err) throw err;
    // use stat
    fs.close(fd, (err) => { if (err) throw err; });
  });
});
```

- La mayoría de los sistemas operativos limitan el número de descriptores de archivo que pueden estar abiertos en un momento dado, por lo que es fundamental cerrar el descriptor cuando se completan las operaciones. De lo contrario, se producirá una pérdida de memoria que eventualmente hará que una aplicación se bloquee.

Lectura y escritura de ficheros

- Para leer un fichero:

```
fs.readFile('message.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

- Para escribir un fichero:

```
fs.writeFile('message.txt', data, 'utf8', (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

API fs.promises

- Cuando hay que realizar un tratamiento de ficheros que requieren una serie de operaciones sucesivas, el anidamiento de los callback puede complicar mucho el proceso.
- El uso de las versiones síncronas no es recomendable por el rendimiento.
- Como alternativa surgen las promesas, mediante el uso de asyc/await se puede utilizar las versiones asíncronas con un estilo secuencial de programación.
- La API fs.promises proporciona un conjunto alternativo de métodos del sistema de archivos asíncronos que devuelven objetos Promise en lugar de usar devoluciones de llamada. Se puede acceder a la API a través de require('fs').promises o directamente con require('fs/promises').

```
const fs = require('fs/promises');

async function copia(fIn, fOut) {
   let datos = await fs.readFile(fIn, 'utf8');
   console.log(datos);
   await fs.writeFile(fOut, datos, 'utf8');
   console.log('Copiado');
}

copia('in.txt', 'out.txt');
```


Formidable

- Módulo Node.js para analizar datos de formularios, especialmente subidas de archivos.

```
$ npm install formidable
var formidable = require("formidable");
app.use('/files', express.static('uploads'))
app.get('/fileupload', function (req, res) {
  res.status(200).end(plantillaHTML('fileupload', `
    <form action="fileupload" method="post" enctype="multipart/form-data">
      <input type="file" name="filetoupload"><input type="submit">
    </form>
  `))
})
app.post('/fileupload', function (req, res) {
  let form = new formidable.IncomingForm();
  form.parse(req, function (err, fields, files) {
    let newpath = __dirname + "/uploads/" + files.filetoupload.name;
    fs.rename(files.filetoupload.path, newpath, function (err) {
      if (err) throw err;
      newpath = "files/" + files.filetoupload.name;
      res.status(200).end(`<a href="${newpath}">${newpath}</a>`);
    });
  });
})
```

Integración con bases de datos

- La adición de la funcionalidad de conectar bases de datos a las aplicaciones se consigue simplemente cargando el controlador de Node.js adecuado para la base de datos en la aplicación.
- Están disponibles módulos de Node.js para los sistemas de base de datos más conocidos, tanto para relacionales como no SQL:
  - Cassandra
  - Couchbase
  - CouchDB
  - ElasticSearch
  - LevelDB
  - MongoDB
  - MySQL
  - Neo4j
  - Oracle
  - PostgreSQL
  - Redis
  - SQL Server
  - SQLite

MySQL

```
$ npm install mysql

var mysql = require('mysql')
var connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'root',
  database: 'sakila'
})

connection.connect()

connection.query('SELECT * FROM `sakila`.`category`', function (err, rows, fields) {
  if (err) throw err

  console.log(rows)
})
```

MongoDB

```
$ npm install mongodb

const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://192.168.99.101:27017';
const dbName = 'microservicios';

const client = new MongoClient(url);

client.connect(function (err) {
   if(err) throw err;
   const db = client.db(dbName);
   const collection = db.collection('personas');
   collection.find({}).toArray(function (err, docs) {
      if(err) throw err;
      console.log(docs);
      client.close();
   });
});
```