Node.js

https://nodejs.org/es/

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 máquina nativo (JIT) 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++.
- El ecosistema Node.js es enorme y gracias a V8, que también impulsa las aplicaciones de escritorio, con proyectos como Electron.

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, sin 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.

Diferencias con el navegador

- Tanto el navegador como Node.js utilizan JavaScript como lenguaje de programación. Crear aplicaciones que se ejecutan en el navegador es algo completamente diferente a crear una aplicación Node.js. A pesar de que siempre es JavaScript, existen algunas diferencias clave que hacen que la experiencia sea radicalmente diferente.
- Desde la perspectiva de un desarrollador frontend que utiliza ampliamente JavaScript, las aplicaciones Node.js traen consigo una gran ventaja: la comodidad de programar todo, el frontend y el backend, en un solo lenguaje.
- En el navegador, la mayor parte del tiempo lo que hace es interactuar con el DOM u otras API de plataforma web, como las cookies. Esos no existen en Node.js, por supuesto, no tiene ni document, window, ni todos los demás objetos que proporciona el navegador. Y en el navegador, no tenemos todas las buenas API que Node.js proporciona a través de sus módulos, como la funcionalidad de acceso al sistema de archivos.
- Otra gran diferencia es que en Node.js se controla el entorno, sabrás en qué versión de Node.js se ejecutará la aplicación. En comparación con el entorno del navegador, donde no puedes darte el lujo de elegir qué navegador usarán tus visitantes, esto es muy conveniente. Esto significa que se puede escribir todo el JavaScript ES2015+ moderno que admita la versión de Node.js. Dado que JavaScript se mueve muy rápido, pero los navegadores y usuarios pueden tardar un poco en actualizarse, a veces en la web uno se ve obligado a utilizar versiones anteriores de JavaScript/ECMAScript o usar Babel para transformar el código para que sea compatible con ES5 antes de enviarlo al navegador, pero en Node.js no es necesario.
- Otra diferencia es que Node.js admite los sistemas de módulos CommonJS y ES (desde Node.js v12), mientras que en el navegador estamos empezando a ver que se implementa el estándar de módulos ES.

HERRAMIENTAS DE DESARROLLO

IDEs

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

Extensiones Visual Studio Code

- Spanish Language Pack for Visual Studio Code
- Code Spell Checker
  - Spanish - Code Spell Checker
- Node Essentials
- REST Client
- OpenAPI (Swagger) Editor
- Path Intellisense

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 en modo global (se puede usar localmente):
  - npm install -g nodemon
- Para ejecutar el servidor en modo monitorizado:
  - nodemon server.js
- Node.js, en la versión 18, incorporo en modo experimental el modo observación:
  - node --watch index.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

ESLint

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- ESLint (https://eslint.org/) es una herramienta para identificar e informar sobre patrones encontrados en código ECMAScript/JavaScript, con el objetivo de hacer que el código sea más consistente y evitar errores: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Se puede instalar ESLint usando npm (las dependencias se instalarán automáticamente) y luego se debe crear un archivo de configuración .eslintrc.json en el directorio:
  - npm install --save-dev eslint eslint-plugin-node
  - npx eslint --init
- Se puede instalar y configurar directamente con:
  - npm init @eslint/config
- Se puede ejecutar ESLint con cualquier archivo o directorio desde línea de comandos o se puede agregar un nuevo script al package.json:
  - npx eslint src/**/*{.js,.jsx,.ts,.tsx}

```
"lint": "eslint src/**/*{.js,.jsx,.ts,.tsx}"
```

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 permitiendo:
  - 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
  - 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 npm:
  - 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.
  - 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 está la ubicación global 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 i
  - 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, -g para las globales:
  - 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.

Lanzamientos Node.js

- Las versiones principales de Node.js entran en el estado de lanzamiento actual durante seis meses, lo que les da a los autores de la biblioteca tiempo para agregarles soporte.
- Después de seis meses, las versiones impares (9, 11, etc.) dejan de ser compatibles y las versiones pares (10, 12, etc.) pasan al estado *LTS activo* y están listas para su uso general.
- El estado de la versión LTS es "soporte a largo plazo", lo que normalmente garantiza que los errores críticos se corregirán durante un total de 30 meses.
- Las aplicaciones de producción solo deben utilizar versiones *Active LTS* o *Maintenance LTS*.

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 minor o en la versión del parche, pero no en la versión principal (major). 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 minor 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 package.json se puede establecer a qué versiones se desea actualizar (parche o menor). El proceso de npm install es asíncrono, descarga las dependencias según van llegando, dependiendo del momento, unas dependencias anidadas adelantan a otras fijando las versiones. Esto provocaba, al realizar el npm install, que el proyecto original y una versión del proyecto recién descargada recibieran diferentes versiones de las dependencias anidadas lo que puede provocar problemas.
- 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 el problema que package.json dejó sin resolver.
- package-lock.json establece "en piedra" la versión instalada actualmente de cada paquete y npm, si existe el package-lock.json, usará esas versiones exactas en el install.
- Para limpiar las dependencias:
  - npm ci
  - npm install-clean

Auditorias de seguridad

- El comando npm audit realiza un examen completo del proyecto para detectar posibles vulnerabilidades de seguridad. A continuación, genera un informe detallado que resalta los problemas identificados. La realización de auditorías de seguridad es un paso fundamental para identificar y solucionar vulnerabilidades dentro de las cadenas de dependencias del proyecto. Solucionar estas vulnerabilidades puede ayudar a evitar problemas como la pérdida de datos, las interrupciones del servicio y el acceso no autorizado a la información confidencial.
- El comando de auditoría envía una descripción de las dependencias configuradas en su proyecto a su registro predeterminado y solicita un informe de vulnerabilidades conocidas. Si se encuentra alguna vulnerabilidad, se calculará el impacto y la solución adecuada.
  - npm audit *fix*
- Si se proporciona el argumento fix, se aplicarán correcciones al árbol de paquetes. Hay que tener en cuenta que algunas vulnerabilidades no se pueden solucionar automáticamente y requerirán intervención o revisión manual. Si el problema reside en una dependencia indirecta, es decir, es un problema de un paquete que utiliza internamente una dependencia directa o alguna de sus dependencias, cambiar sus rangos de dependencia puede tener los efectos secundarios desafortunados. Se puede forzar la actualización con:
  - npm audit fix --force

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.

Publicación

- Aunque existen numerosas estrategias para publicar y distribuir paquetes (módulos, bibliotecas de componentes, …), es altamente recomendado publicarlos en NPM. El repositorio npmjs.com es un registro de software en línea para compartir bibliotecas, herramientas, utilidades, paquetes, etc.
- Una vez que la biblioteca se publica en NPM, otros proyectos pueden agregar el paquete como una dependencia y usar el contenido dentro del propio proyecto.
- Para poder publicar en NPM es necesario disponer de una cuenta de usuario: gratuita (solo paquetes públicos) o Pro/Teams (también paquetes privados).
- Se debe iniciar sesión desde línea de comandos:
  - npm login
- El inicio de sesión es global. Para consultar el usuario actual:
  - npm whoami
- La sesión se mantiene abierta hasta que no se cierre explícitamente:
  - npm logout

Publicación

- Como paso previo quizás sea necesario renombrar el paquete dado que el nombre del paquete tiene que cumplir ciertas reglas para ser válido: el nombre debe ser único en el repositorio, debe tener un máximo de 214 caracteres, no puede llevar mayúsculas ni espacios, no puede empezar por un punto o un guion bajo, no puede contener caracteres que no sean seguros para URL…
- El archivo package.json puede contener información adicional para su publicación en el repositorio de NPM: author, homepage, license, keywords, repository, engines, browserslist, bugs …
- El archivo README.md puede incluir instrucciones para instalar, configurar y usar el código en su paquete, así como cualquier otra información que un usuario pueda encontrar útil.
- El archivo .npmignore, similar al .gitignore, permite excluir archivos y carpetas para los paquetes generados por npm para publicar. Para generar un archivo .tgz y poder verificar lo que se subirá (excluir o borrar una consultado):
  - npm pack
- Para publicar (se enviará directamente al registro npm), una ver probado y verificado, actualizado el número de versión:
  - npm publish --access public
- Para eliminar un paquete del registro (sin restricciones las primeras de 72 horas):
  - npm unpublish <package_name>@<version>

package.json

```
{
  "name": "jma-calculadora",
  "version": "1.0.0",
  "description": "Componente demo del curso",
  "main": "dist/index.cjs.js",
  "module": "dist/index.js",
  "es2015": "dist/esm/index.mjs",
  "es2017": "dist/esm/index.mjs",
  "types": "dist/types/index.d.ts",
  "collection": "dist/collection/collection-manifest.json",
  "collection:main": "dist/collection/index.js",
  "unpkg": "dist/jma-calculadora/jma-calculadora.esm.js",
  "keywords": ["curso", "demos", "stencil", "web component"],
  "homepage": "https://github.com/ionic-team/stencil-component-starter",
  "repository": { "type": "git", "url": "https://github.com/ionic-team/stencil-component-starter.git" },
  "files": [ "dist/", "loader/" ],
```


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.
- 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 sintaxis asyc/await permite un estilo de programación secuencial en procesos asíncronos, delegando en el compilador o interprete su implementación.
- La declaración de función asyc define una función asíncrona, que devuelve un objeto AsyncFunction. Una función asíncrona 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() {
  let x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}
```

EcmaScript 2016 (ES7)

- El operador exponencial (**):
  - x = Math.pow(3, 2);
  - x = 3 ** 2;
  - x = 3; x **= 2;
- Método Array.prototype.includes()
  - if(listado.indexOf(item) !== -1) // lo contiene
  - if(listado.includes(item)) // lo contiene

EcmaScript 2017 (ES8)

- Object.values(): devuelve un array con los valores correspondientes a las propiedades enumerables de un objeto.
- Object.entries(): devuelve una matriz de arrays [key, value] del objeto dado.
- Object.getOwnPropertyDescriptors(): devuelve todos los descriptores de propiedad propios de un objeto dado para su clonado.
  - value: El valor asociado con la propiedad (solo descriptores de datos).
  - writable: true si y solo si el valor asociado con la propiedad puede ser cambiado (solo descriptores de datos).
  - get: Un función que sirve como un getter para la propiedad, o undefined si no hay getter (solo descriptores de acceso).
  - set: Una función que sirve como un setter para la propiedad, o undefined si no hay setter (solo descriptores de acceso).
  - configurable: true si y solo si el tipo de este descriptor de propiedad puede ser cambiado y si la propiedad puede ser borrada de el objeto correspondiente.
  - enumerable: true si y solo si esta propiedad aparece durante la enumeración de las propiedad en el objeto correspondiente.
- str.padStart() y str.padEnd() rellenan la cadena actual, por el principio o por el final, con la cadena dada (repitiéndola si es necesaria) para que la cadena resultante alcance una longitud dada, si aun no la tiene.
- Ahora permite tener comas finales después del último parámetro de función.

Memoria compartida con acceso atómico (ES8)

- Cuando dos threads tienen acceso a un espacio de memoria compartido, es importante que las operaciones de escritura o lectura sean atómicas, es decir, que se hagan en bloque y aisladas de otros threads, para garantizar la validez de los datos.
- Esta funcionalidad está pensada para web workers, ya que es el único caso en el se puede implementar multithreading en Javascript, y de momento está orientada a muy bajo nivel: ArrayBuffers (arrays de datos binarios).
- El mecanismo consta de:
  - SharedArrayBuffer: Un nuevo constructor para crear ArrayBuffers de memoria compartida.
  - La clase Atomic, con métodos estáticos para acceder/manipular ese SharedArrayBuffer de forma atómica.

EcmaScript 2018 (ES9)

- ES6 incorporaba el spread operator y los parámetros rest para hacer asignaciones por destructuring con Arrays.

```
const primes = [2, 3, 5, 7, 11];
const [first, second, ...rest] = primes;
console.log(rest); // [5, 7, 11]
const primes2 = [first, second, ...rest];
console.log(primes2); // [2, 3, 5, 7, 11]
```

- ES9 permite hacer asignaciones por destructuring con objetos.

```
const person = { firstName: 'Peter', lastName: 'Parker', age: 26, }
const { age, ...name } = person;
console.log(name); // {firstName: "Peter", lastName: "Parker"}
const person2 = { age, ...name };
console.log(person2); // {age: 26, firstName: "Peter", lastName: "Parker"}
```

- promesa.finally: permite llamar siempre a una función de callback sin parámetros al completarse (resuelta o rechazada).
- For asíncrono: for await (const line of readLines(filePath)) {

EcmaScript 2018 (ES9)

- Mejoras en las expresiones regulares:
  - Named capture groups

```
let re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
let result = re.exec('2015-01-02');
// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02’;
```

  - Comprobar si la expresión esta precedida de un determinado patrón sin incluirla en el resultado:

```
const regex = /(?<=€)([0-9])+/;
let result = regex.test('€2000’); // result === 2000
```

  - El indicado /s evita los problemas con las secuencias de escape del . como comodín:

```
/hola.mundo/.test('hola\tmundo'); //false
/hola.mundo/s.test('hola\tmundo'); //true
```


EcmaScript 2019 (ES10)

- Object.fromEntries(): transforma una lista de arrays [clave-valor] en un objeto, proceso contrario al método entries()

```
const arr = [["firstName", "Peter"], ["lastName", "Parker"], ["age", 26]];
Object.fromEntries(arr) // = { firstName: 'Peter', lastName: 'Parker', age: 26 }
```

- Array.prototype.flat(): permite aplanar un array multidimensional por niveles.

```
const arr = ['1', '2', ['3.1', '3.2'], '4', ['5.1', ['5.2.1', '5.2.2']], '6'];
arr.flat();      // ["1", "2", "3.1", "3.2", "4", "5.1", Array(2), "6"]
arr.flat(2);     // ["1", "2", "3.1", "3.2", "4", "5.1", "5.2.1", "5.2.2", "6"]
```

- Array.prototype.flatMap(): combina el método map(callback) con el flat() para transformar y aplanar un array de forma eficiente.

```
const arr = [[1, 2], [3, 4], [5, 6]];
arr.map(([a, b]) => [a, a * b]) // [[1, 2], [3, 12], [5, 30]]
   .flat() // [1, 2, 3, 12, 5, 30]
arr.flatMap(([a, b]) => [a, a * b]) // [1, 2, 3, 12, 5, 30]
```

EcmaScript 2019 (ES10)

- String.trimStart(), String.trimEnd(): eliminan los espacios en blanco al principio o del final de un string. Por consistencia con su proceso contrario padStart y padEnd, se ha mantenido la nomenclatura pero se añaden también como alias trimLeft y trimRight por compatibilidad con otros.
- Symbol.description: Obtiene la cadena asociada al símbolo.
- Function.prototype.toString() ahora devuelve la cadena con el código fuente sin eliminar espacios en blanco, comentarios y saltos de línea.
- La captura del error en una referencia al declarar el catch ahora es opcional:

```
try {
   :
} catch {    // catch(e)
   :
}
```

- JSON.stringify(), independiente del formato de entrada, debe devolver una cadena UTF-8 bien formada y JSON.parse() debe aceptar los símbolos de separador de línea (\u2028) y separador de párrafo (\u2029).

ECMAScript 2020 (ES11)

- Operadores:
  - Encadenamiento opcional, cortocircuitado (?.):
    - v?.p → v === null ? null : v.p
  - Selección de valores (||):
    - v || "default!" → v ? v : "default!"
  - Coalescencia nula (??):
    - v ?? "default!" → (v == null || v == undefined) ? "default!" : v
- El objeto globalthis establece una forma universal de acceder al objeto global (this) en JavaScript que anteriormente dependía del entorno, usando window o self en el navegador, o global en NodeJs.
- Ahora el orden de las propiedades está garantizado en el bucle for in o JSON.stringify.

ECMAScript 2020 (ES11)

- Aparece el BigInt para superar algunas limitaciones de (2^{53}) de ciertas APIS cuando los ids numéricos son muy grandes (los ids usados por Twitter), y cuyo valor superaba el valor de Number.MAX_SAFE_INTEGER, obligando a tener que tratarlos como strings. Se pueden crear números enteros más grandes añadiendo una "n" al final del entero o mediante la función BigInt.
- Strings.matchAll(): genera un iterador con todos los objetos coincidentes generados por una expresión regular global.
- Promise.allSettled: Ejecuta una colección de promesas y devuelve el resultado de cada una de ellas para su tratamiento como resuelta o rechazada.

ECMAScript 2020 (ES11)

- Importación dinámica de módulos como promesas:

```
import(`./my-modules/${ myModuleName }.js`)
  .then(module => {  module.doStuff(); })
  .catch(err => console.log(err));
(async () => {
  const module = await import (`./my-modules/${myModuleName }.js`);
  module2.doStuff();
})();
```

- El objeto import.meta expone el contenido especifico con los metadatos de módulo JavaScript.

```
<script type="module" src="my-module.js"></script>
console.log(import.meta); // { url: "file:///home/user/my-module.js" }
```

- Propagación (importación y exportación):

```
export * as ns from 'module'
```

ECMAScript 2021 (ES12)

- Separador numérico: permite separar los números con guiones bajo (_) sin afectar al valor (azúcar sintáctica): 100_000_000 === 100000000
- Operador de asignación lógica: se incorporan los operadores de asignación lógica &&= ||= y ??=
- Métodos, getters y setters privados en las clases: con el prefijo # en su nobre verán su alcance limitado a la clase donde están definidos.

```
#private(val) { … }
set #width  (width)  { this.#width = width; }
get width  ()      { return this.#width ; }
```

- String.replaceAll() reemplaza todas las ocurrencias de una cadena por otra cadena (replace() solo reemplazaba la primera ocurrencia).
- Promise.any(), cuyo argumento es un array de promesas, será gestionado capturando la respuesta de la primera que sea resuelta de forma satisfactoria, o un array de excepciones en caso de que todas sean rechazadas

```
Promise.any([promise1, promise2, promise3])
```


ECMAScript 2022 (ES13)

- Await a nivel superior: permite usar el await sin necesidad de estar contenido dentro de una función asíncrona (marcada con async).
- Object.hasOwn(): devuelve true si el objeto especificado tiene la propiedad indicada como propiedad propia. Si la propiedad es heredada, o no existe, el método devuelve false.
- Array.at(): recibe un valor numérico entero y devuelve el elemento en esa posición, permitiendo valores positivos y negativos. Los valores negativos contarán desde el último elemento del array.

```
[1,2,3,4,5].at(-1)   // devuelve 5
```

- error.cause: permite especificar que causó un error.

```
try {
  connectToDatabase();
} catch (err) {
  throw new Error("Connecting to database failed.", { cause: err });
}
```

ECMAScript 2022 (ES13)

- Propiedades y métodos privados: Se pueden crear atributos y métodos privados en la clases usando el prefijo # en el nombre.

```
class Usuario{
  #nombre;
  constructor(nombre) {
    this.#nombre = nombre;
  }
  #apellidos(apellido) {
    return this.#nombre + ' ' + apellido;
  }
  mostrar(apellido) {
    console.log(this.#apellidos(apellido));
  }
}

const usuario = new Usuario("Pepito");
console.log(usuario.#nombre);                // Error
console.log(usuario.#apellidos("Grillo")); // Error
console.log(usuario.mostrar("Grillo"));     // Pepito Grillo
```


TYPESCRIPT

Introducción

- TypeScript es un lenguaje de código abierto moderno mantenido y desarrollado por Microsoft. Es amado y utilizado por muchos desarrolladores de software de todo el mundo. Básicamente, es un superconjunto de JavaScript que agrega nuevas capacidades al lenguaje.
- La adición más notable son las definiciones de tipos estáticos, algo que no está presente en JavaScript simple. Gracias a los tipos es posible, por ejemplo, declarar qué tipo de argumentos esperamos y qué se devuelve exactamente en nuestras funciones o cuál es la forma exacta del objeto que estamos creando.
- TypeScript ofrece muchos otros mecanismos excelentes, como interfaces, clases, tipos de utilidades, etc. Además, en proyectos más grandes, puede declarar la configuración del compilador TypeScript en un archivo separado y ajustar detalladamente cómo funciona, qué tan estricto es y dónde almacena los archivos compilados, por ejemplo. Se puede adoptar progresivamente, ayuda a que el código sea más legible y comprensible y permite a los desarrolladores utilizar funciones de lenguaje moderno mientras envían código para versiones anteriores de Node.js.
- TypeScript es una herramienta realmente poderosa y abre un nuevo mundo de posibilidades en proyectos de JavaScript. Hace que nuestro código sea más seguro y robusto al evitar muchos errores incluso antes de que se envíe el código; detecta problemas durante el desarrollo del código y se integra maravillosamente con editores de código como Visual Studio Code.

Node.JS

- Crear aplicación Node
  - mkdir node-ts && cd node-ts
  - mkdir src
  - npm init -y
- Instalar dependencias
  - npm i --save-dev typescript @types/node ts-node nodemon rimraf
- Configurar nodemon creando el fichero nodemon.json
- Configurar TypeScript en tsconfig.json
  - npx tsc --init
- Configurar los scripts en package.json
- Crear las aplicación en src/index.ts
- Ejecutar en modo desarrollo:
  - npm run start:dev

```
import http from 'node:http';
function saluda(nombre: string): string {
    return `Hola ${nombre}!`;
}
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(saluda('Mundo'));
});
server.listen(8080, '127.0.0.1');
```

Node.JS

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "allowJs": true,
    "rootDir": "src",
    "outDir": "build",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

nodemon.json

```
{
  "watch": ["src"],
  "ext": ".ts,.js",
  "ignore": [],
  "exec": "npx ts-node ./src/index.ts"
}
```

package.json

```
"main": "build/index.js",
"scripts": {
  "build": "rimraf ./build && tsc",
  "start:dev": "npx nodemon",
  "start": "npm run build && node build/index.js",
```


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 (nombre del directorio del módulo actual), __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();
require('dotenv').config({ path: './.env.local', override: true, debug: true })
```


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}`);
});
```

Desarrollo y Producción

- Se pueden tener diferentes configuraciones para entornos de producción y desarrollo.
- Node.js supone que siempre se ejecuta en un entorno de desarrollo. Se puede indicar a Node.js que se está ejecutando en producción configurando la variable de entorno a NODE_ENV=production. Esta variable de entorno es una convención que también usan ampliamente las bibliotecas externas.
- Esto generalmente se hace ejecutando el comando:
  - export NODE_ENV=production (bash)
  - set NODE_ENV=production (Windows)
- Establecer el entorno en productiongeneral garantiza que
  - El registro se mantiene a un nivel mínimo y esencial
  - Se realizan más niveles de almacenamiento en caché para optimizar el rendimiento.
- Se puede utilizar declaraciones condicionales para ejecutar código en diferentes entornos:

```
if (process.env.NODE_ENV === 'development') {
  // ...
} else if (process.env.NODE_ENV === 'production') {
  // ...
}
if (['production', 'staging'].includes(process.env.NODE_ENV)) {
  // ...
}
```


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:

```
const 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 entrada 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 desde 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 desde archivos diferentes.

Módulos ES2015

- Ahora Node.js tiene dos sistemas de módulos: módulos CommonJS y módulos ECMAScript. Node.js está adoptando progresivamente los módulos ECMAScript, no todos los módulos se pueden importar.
- Una declaración import puede hacer referencia a un módulo ES o un módulo CommonJS. Las declaraciones import solo se permiten en módulos ES, pero las expresiones dinámicas import() se admiten en CommonJS para cargar módulos ES.

```
import fs, { readFileSync } from 'node:fs';
```

- Al importar módulos CommonJS, el objeto module.exports se proporciona como exportación predeterminada. Es posible que haya exportaciones nombradas disponibles, proporcionadas por análisis estático.
- Los módulos ECMAScript no tienen:
  - require, require.resolve, require.extensions, require.cache, exports o module.exports
  - __filename o __dirname (se pueden replicar mediante import.meta.url)
  - los complementos no son compatibles con las importaciones de módulos ES.
  - NODE_PATH no forma parte de la resolución de especificadores import .

Control de flujo asíncrono

- En esencia, JavaScript está diseñado para no bloquear el hilo "principal", aquí es donde se representan las vistas. Esto es fundamental en el navegador: cuando el hilo principal se bloquea, se produce el infame "congelamiento" que temen los usuarios finales, y no se pueden enviar otros eventos, lo que resulta en la pérdida de adquisición de datos, por ejemplo.
- Esto crea algunas limitaciones únicas que sólo un estilo funcional de devoluciones de llamada puede solucionar.
- Una devolución de llamada es una función simple que se pasa como un argumento a otra función y solo se ejecutará cuando ocurra el evento. Podemos hacer esto porque en JavaScript las funciones son objetos, que pueden asignarse a variables y pasarse a otras funciones (denominadas funciones de orden superior).

Control de flujo asíncrono

- Las devoluciones de llamada son excelentes para casos simples. Sin embargo, las devoluciones de llamada pueden resultar difíciles de manejar en procedimientos más complicados.
- Cada devolución de llamada agrega un nivel de anidamiento y, cuando tienes muchas devoluciones de llamada, el código comienza a complicarse muy rápidamente. Esto a menudo resulta en un "infierno de devoluciones de llamadas", donde múltiples funciones anidadas con devoluciones de llamadas hacen que el código sea más difícil de leer, depurar, organizar, etc.
- A partir de ES2015, JavaScript introdujo varias funciones que nos ayudan con código asincrónico que no implica el uso de devoluciones de llamada: Promises (ES2015) y Async/Await (ES2017).

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.

Diseño sin bloqueos

- La ejecución de JavaScript en Node.js en un solo subproceso, el bucle de eventos y el control de flujo asíncrono sin bloqueo implican una serie de desafíos de diseño propios de Node.js.
- Para solventar dichos desafíos, Node.js suministra los siguientes recursos:
  - Temporizadores
  - Eventos
  - Streams
  - Buffer
  - Tratamientos asíncronos de errores
  - Soporte de promesas

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 del 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.

Temporizadores

- 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().

process.nextTick()

- Cada vez que el bucle de eventos realiza un recorrido completo, lo llamamos tic. Cuando pasamos una función a process.nextTick(), le indicamos al motor que invoque esta función al final de la operación actual, antes de que comience el siguiente ciclo de eventos:

```
process.nextTick(() => {
  // do something
});
```

- El bucle de eventos está ocupado procesando el código de función actual. Cuando finaliza esta operación, el motor JS ejecuta todas las funciones pasadas en las llamadas nextTick durante esa operación. Es la forma en que podemos decirle al motor JS que procese una función de forma asincrónica (después de la función actual), pero lo antes posible, sin ponerla en cola.
- La llamada setTimeout(() => {}, 0) ejecutará la función al final del siguiente tick, mucho más tarde que cuando se use nextTick(), lo que prioriza la llamada y la ejecuta justo antes del comienzo del siguiente tick. Se usa nextTick() cuando se desee asegurarse de que en la siguiente iteración del bucle de eventos ese código ya esté ejecutado.

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, llamados "listeners", "oyentes" o "controladores de eventos".
- 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 camellCase, 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(), pero requieren haber cacheado 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 (se elimina automáticamente la primera vez que se ejecuta).
- 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 las canalizaciones 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 adecuada y optimizada 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 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.

Errores

- 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
}
```

- Los errores que ocurren dentro de las API asincrónicas pueden informarse y tratarse de varias maneras.
- La mayoría de los métodos asíncronos expuestos por la API principal de Node.js siguen un patrón del lenguaje 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 es una instancia de Error, no es null, entonces ha ocurrido un error que debería ser tratado.

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


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

- A partir de ES2015, JavaScript introdujo varias funciones que nos ayudan con código asincrónico que no implica el uso de devoluciones de llamada: Promises (ES2015) y Async/Await (ES2017).
- La sintaxis asyc/await permite un estilo de programación secuencial en procesos asíncronos, delegando en el compilador o interprete su implementación.
- El operador await se usa para esperar a una Promise y sólo dentro de una async function.
- 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.

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();
```

Módulos del API

- Assert
- Async hooks
- Asynchronous context tracking
- Buffer
- Child process
- Cluster
- Console
- Crypto
- Debugger
- DNS
- Domain
- Errors
- Events
- File system
- HTTP
- HTTP2
- HTTPS
- Inspector
- Net
- OS
- Packages
- Path
- Performance measurement APIs
- Permissions
- Policies
- Process
- Punycode
- Query string
- Readline
- REPL
- Stream
- String decoder
- Test runner
- Timers
- TLS
- Trace events
- TTY
- UDPdatagram sockets
- URL
- Usage and example
- Util
- V8
- VM
- Web Crypto API
- Web Streams API
- WebAssembly System Interface
- Worker threads
- Zlib

Node.js Frameworks

- Los marcos de Node.js se han convertido en la mejor opción para crear aplicaciones web, de escritorio y móviles o microservicios que sean interactivas, rentables, escalables y orientadas a objetivos.
  - Express.js
  - Koa.js
  - Fastify.js
  - Nest.js
  - Next.js
  - Nuxt.js
  - Sails.js
  - Adonis.js
  - Total.js
  - LoopBack.js
  - Restify.js
  - Hapi.js
  - tinyhttp.js
  - Socket.IO
  - Apollo Server
  - Meteor.js
  - Electron
  - React Native
  - Ionic
  - NativeScript

Hypertext Transfer Protocol

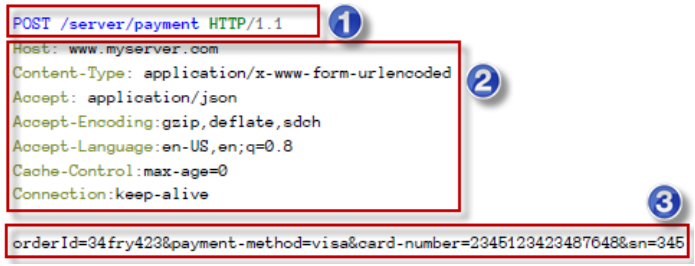
HTTP

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.

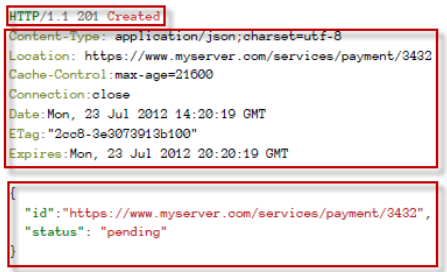
Petición HTTP

- 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 la cabecera del 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 en la petición

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 en la petición

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

Encabezado HTTP Cache-Control

- El encabezado HTTP Cache-Control especifica directivas (instrucciones) para almacenar temporalmente (caching) tanto en peticiones como en respuestas. Una directiva dada en una petición no significa que la misma directiva estar en la respuesta.
- Los valores estándar que pueden ser usados por el servidor en una respuesta HTTP son:
  - public: La respuesta puede estar almacenada en cualquier memoria cache.
  - private: La respuesta puede estar almacenada sólo por el cache de un navegador.
  - no-cache: La respuesta puede estar almacenada en cualquier memoria cache pero DEBE pasar siempre por una validación con el servidor de origen antes de utilizarse.
  - no-store: La respuesta puede no ser almacenada en cualquier cache.
  - max-age=<seconds>: La cantidad máxima de tiempo un recurso es considerado reciente.
  - s-maxage=<seconds>: Anula el encabezado max-age o el Expires, pero solo para caches compartidos (e.g., proxies).
  - must-revalidate: Indica que una vez un recurso se vuelve obsoleto, el cache no debe usar su copia obsoleta sin validar correctamente en el servidor de origen.
  - proxy-revalidate: Similar a must-revalidate, pero solo para caches compartidos (es decir, proxies). Ignorado por caches privados.
  - no-transform: No deberían hacerse transformaciones o conversiones al recurso.

Encabezados HTTP ETag, If-Match y If-None-Match

- El encabezado de respuesta de HTTP ETag es un identificador (resumen hash) para una versión específica de un recurso y los encabezados If-Match e If-None-Match de la solicitud HTTP hace que la solicitud sea condicional.
- Para los métodos GET y HEAD con If-None-Match: si el ETag no coincide con los dados, el servidor devolverá el recurso solicitado con un estado 200, si coincide el servidor debe devolver el código de estado HTTP 304 (No modificado) y DEBE generar cualquiera de los siguientes campos de encabezado que se habrían enviado en una respuesta 200 (OK) a la misma solicitud: Cache-Control, Content-Location, Date, ETag, Expires y Vary.
- Para los métodos PUT y DELETE con If-Match: si el ETag coincide con los dados, se realiza la actualización o borrado y se devuelve un estado HTTP 204 (sin contenido) incluyendo el Cache-Control y el ETag de la versión actualizada del recurso en el PUT. Si no coinciden, se ha producido un error de concurrencia, la versión del servidor ha sido modificada desde que la recibió el cliente, debe devolver una respuesta HTTP con un cuerpo de mensaje vacío y un código de estado 412 (Precondición fallida).
- Si los datos solicitados ya no existen, el servido debe devolver una respuesta HTTP con el código de estado 404 (no encontrado).

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, …) que 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 en el directorio actual.
  - md myapp & cd myapp & npx express-generator --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
  - javascripts
  - images
  - 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('hola') },
  'text/html': function () { res.send('<html><body><p>hola</p></body></html>') },
  'application/json': function () { res.json({ message: 'hola' }) },
  'default': function () { res.status(406).send('Not Acceptable') }
})
```


XML

- Dependencias:
  - npm install express-xml-bodyparser xml2js
- Middleware de análisis del cuerpo

```
const xmlParser = require('express-xml-bodyparser');
app.use(xmlParser());
```

- Serialización del body:

```
const xml2js = require('xml2js');
const builder = new xml2js.Builder();
res.status(200).end(builder.buildObject(data))
```

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 en una instancia del objeto de la aplicación se utiliza las funciones app.use() y app.METHOD(), donde METHOD es 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 y se deben proporcionar los cuatro argumentos para que sea identificado como una función de manejo de errores. Incluso si no se necesita usar el objeto next, se debe especificar 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.

Registro de solicitudes HTTP

- Instalación de los módulos:
  - npm install morgan rotating-file-stream
- Para activar el registro como un middleware de Espress con un formato predefinido y un fichero rotatorio:

```
const logger = require('morgan')
const rfs = require('rotating-file-stream')

app.use(logger('combined', {
  stream: rfs.createStream("file.log", {
    path: path.join(__dirname, 'log'), // directory
    size: "10M", // rotate every 10 MegaBytes written
    interval: "1d", // rotate daily
    compress: "gzip" // compress rotated files
  })
}))
```

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 !{'<span>' + title + '</span>’}
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 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');
```


Multer

- Multer es un "middleware" de node.js para el manejo de multipart/form-data, el cuál es usado sobre todo para la subida de archivos.
- Instalación
  - $ npm install multer
- Crear un directorio de subida (ej: uploads)
- Configurar para que guarde los ficheros en el directorio de subida, con el nombre original del fichero y un limite máximo de 2 mb por fichero:

```
const upload = multer({
  limits: { fileSize: 2 * 1024 * 1024 /* 2mb */},
  storage: multer.diskStorage({
    destination: (req, file, cb) => cb(null, 'uploads/'),
    filename: (req, file, cb) => cb(null, file.originalname)
  })
})
```

Multer

```
app.use('/files', express.static('uploads'))
app.get('/fileupload', function (req, res) {
 res.status(200).end(`
  <html><body><h1>Multiple file uploads</h1>
  <form action="fileupload" method="post" enctype="multipart/form-data">
     <input type="file" name="filestoupload" multiple="multiple" required><input type="submit">
  </form>
  </body></html>`)
})
app.post('/fileupload', upload.array('filestoupload'), async (req, res, next) => {
 let rutas = req.files.map(f => ({ url: `${req.protocol}://${req.headers.host}/files/${f.originalname}` }))
 if (req.headers?.accept?.includes('application/json'))
  res.status(200).json(rutas).end();
 else {
  res.status(200).end(`
  <html><body><h1>Uploads</h1>
  <ul>${rutas.map(r => `<li><a href="${r.url}">${r.url}</a></li>`).join('')}</ul>
  </body></html>`);
 }
})
```


BASES DE DATOS

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

Instalación con Docker

- Docker Toolbox
  - Windows 10 Pro ++: https://docs.docker.com/docker-for-windows/install/
  - Otras: https://github.com/docker/toolbox/releases
- Ejecutar Docker QuickStart
- Para crear el contenedor de MySQL con la base de datos Sakila:
  - docker run -d --name mysql-sakila -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 jamarton/mysql-sakila
- Para crear el contenedor de MongoDB:
  - docker run -d --name mongodb -p 27017:27017 mongo
- Para crear el contenedor de Redis:
  - docker run --name redis -p 6379:6379 -d redis
  - docker run -d --name redis-commander -p 8081:8081 rediscommander/redis-commander:latest

Instalación

- Instalar SQLite para la base de datos SQLite
  - npm install sqlite3
- Instalar MySQL para la base de datos MySQL
  - npm install mysql
  - npm install mysql2
- Instalar MariaDB para la base de datos MariaDB
  - npm install mariadb
- Instale PostgreSQL (y hstore) para la base de datos PostgreSQL
  - npm install pg pg-hstore
- Instale MSSQL (y controlador tedious) para Microsoft SQL Server
  - npm install tedious

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)
})
```

SQL Server

```
$ npm install tedious

const { Connection, Request } = require('tedious')
function executeStatement() {
  const request = new Request(`SELECT ProductCategoryID, Name FROM SalesLT.ProductCategory WHERE ParentProductCategoryID IS NULL`, (err) => {
    if (err) { console.log(err); }
  });
  request.on('row', (columns) => { console.log(columns.map(col => (col.value ?? 'NULL')).join(' ')); });
  request.on('done', (rowCount, more) => { console.log(rowCount + ' rows returned'); });
  request.on("requestCompleted", (rowCount, more) => { connection.close(); });
  connection.execSql(request);
}
const connection = new Connection({
  server: 'localhost',
  authentication: { type: 'default', options: { userName: 'sa', password: 'P@$$w0rd' } },
  options: { encrypt: false, database: 'AdventureWorksLT2022' }
});
connection.on('connect', (err) => {
  console.log("Connected");
  executeStatement();
});

connection.connect();
```

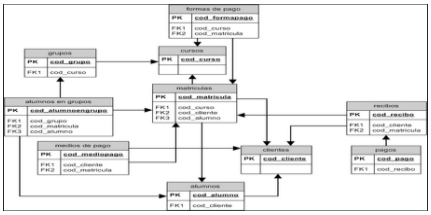

MongoDB

```
$ npm install mongodb

const { MongoClient } = require('mongodb');

const url = 'mongodb://localhost:27017';
const dbName = 'microservicios';
const client = new MongoClient(url);
(async () => {
   try {
      const db = client.db(dbName);
      const collection = db.collection('contactos');
      let docs = await collection.find({}).toArray()
      console.log(docs);
   } catch (error) {
      console.error(error);

   } finally {
      client.close();
   }
})()
```

Discrepancia del Paradigma

- Cuando trabajamos con diferentes paradigmas (Objetos vs Relacional) la discrepancia a la hora de unirlos surge a medida que complicamos el diseño.
- Cuando disponemos de clases sencillas y relaciones sencillas, no hay problemas.

- Lo anterior no es lo habitual. Pero esto SI:



Discrepancia del Paradigma

- Bauer & King (Bauer, C. & King, G. 2007) presentan una lista de los problemas de discrepancia en los paradigmas objeto/relacional
  - Problemas de granularidad
    - O-O pueden definir clases con diferentes niveles de granularidad.
      - Cuanto más fino las clases de grano (Direcciones), éstas pueden ser embebida en las clases de grano grueso (Usuario)
    - En cambio, el sistema de tipo de la base de datos SQL son limitados y la granularidad se puede aplicar sólo en dos niveles
      - en la tabla (tabla de usuario) y el nivel de la columna (columna de dirección)
  - Problemas de subtipos
    - La herencia y el polimorfismo son las características básicas y principales del lenguaje de programación O-O.
    - Los motores de base de datos SQL en general, no son compatibles con subtipos y herencia de tablas.

Discrepancia del Paradigma

- Problemas de identidad
  - Los objetos definen dos nociones diferentes de identidad:
    - Identidad de objeto o referencia (equivalente a la posición de memoria, comprobar con un ==).
    - La igualdad como determinado por la aplicación de los métodos equals ().
  - La identidad de una fila de base de datos se expresa como la clave primaria.
  - Ni equals() ni == es equivalente a la clave principal.
- Problemas de Asociaciones
  - El lenguaje O-O representa a las asociaciones mediante utilizan referencias a objetos
    - Las asociaciones entre objetos son punteros unidireccionales.
    - Relación de pertenencia → Elementos contenidos
    - Modelo de composición (colecciones) → Modelo jerárquico
  - Las asociaciones en BBDD están representados mediante la migración de claves.
    - Todas las asociaciones en una base de datos relacional son bidireccional

ORM

- Las siglas ORM significan “Object-Relational mapping” (Mapeo Objeto-Relacional): técnica de mapear datos desde una representación de modelo de objeto a una representación de modelo de datos relacional (y viceversa).
- Un ORM es un framework de persistencia de nuestros datos (objetos) a una base de datos relacional, es decir, código que escribimos para guardar y recuperar el valor de nuestras clases en una base de datos relacional.
- ORM es el nombre dado a las soluciones automatizadas para solucionar el problema de falta de coincidencia (Objetos-Relacional).
- Un buen número de sistemas de mapeo objeto-relacional se han desarrollado a lo largo de los años, pero su efectividad en el mercado ha sido diversa.

Hibernate	OpenJPA	TopLink	Kodo
CocoBase	DataNucleus	EclipseLink	Amber

ORM

- Todo ORM deben de cumplir 4 especificaciones:
  1. Un API para realizar operaciones básicas CRUD sobre los objetos de las clases persistentes.
  2. Un lenguaje o API para la especificación de las consultas que hacen referencia a las clases y propiedades de clases (SQL).
  3. Un elemento de ORM (SessionFactory) para interactuar con objetos transaccionales y realizar operaciones diversas (conexión, validación, optimización, etc)
  4. Un mecanismo para especificar los metadatos de mapeo (fichero XML, anotaciones)

ORM vs ODM

- ORM (Object Relational Mapper) mapea las relaciones entre los datos, mientras que ODM (Object Document Mapper) se ocupa de mapear los documentos.

- ORM Sequelize
  - https://sequelize.org/
- ODM Mongoose
  - https://mongoosejs.com/

Sequelize

- Sequelize es una herramienta ORM de Node.js basada en promesas para Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, Amazon Redshift y Snowflake's Data Cloud. Cuenta con un sólido soporte de transacciones, relaciones, carga ansiosa y diferida, replicación de lectura y más.
- Instalación
  - npm install sequelize
- Para cargar el paquete:

```
const { Sequelize } = require('sequelize');
```

- Para conectarse a la base de datos, se debe crear una instancia de Sequelize. Esto se puede hacer pasando los parámetros de conexión por separado al constructor de Sequelize o pasando un único URI de conexión:

```
const sequelize = new Sequelize('mysql://root:root@localhost:3306/sakila')
const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: /* 'mysql' | 'mariadb' | 'postgres' | 'mssql' */
});
```


Modelos

- Los modelos son la esencia de Sequelize. Un modelo es una abstracción que representa una tabla en la base de datos. En Sequelize, es una clase que extiende Model.
- El modelo informa a Sequelize sobre la entidad que representa, el nombre de la tabla en la base de datos y qué columnas tiene (y sus tipos de datos).
- Un modelo en Sequelize tiene un nombre. Este nombre no tiene que ser el mismo nombre de la tabla que representa en la base de datos. Por lo general, los modelos tienen nombres en singular (como User) mientras que las tablas tienen nombres en plural (como Users), aunque esto es totalmente configurable.
- Los modelos se pueden definir de dos formas equivalentes en Sequelize:
  - Llamando sequelize.define(modelName, attributes, options)
  - Extendiendo el modelo y llamando init(attributes, options)
- Una vez que se define un modelo, está disponible dentro sequelize.models con el nombre de modelo.
- A través del modelo se interactuara con la base de datos.

sequelize.define

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:');

const User = sequelize.define('User', {
 // Model attributes are defined here
 id: {
   type: DataTypes.INTEGER,
   allowNull: false,
   primaryKey: true,
   autoIncrement: true,
 },
 firstName: {
  type: DataTypes.STRING,
  allowNull: false
 },
 lastName: {
  type: DataTypes.STRING
  // allowNull defaults to true
 }
}, {
 // Other model options go here
});
```


Extendiendo el modelo

```
const { Sequelize, DataTypes, Model } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:');

class User extends Model {}
User.init({ // Model attributes are defined here
 id: {
   type: DataTypes.INTEGER,
   allowNull: false,
   primaryKey: true,
   autoIncrement: true,
 },
 firstName: {
  type: DataTypes.STRING,
  allowNull: false
 },
 lastName: {
  type: DataTypes.STRING
  // allowNull defaults to true
 }
}, { // Other model options go here
 sequelize, // We need to pass the connection instance
 modelName: 'User' // We need to choose the model name
});
```

Atributos y Columnas

- type: cadena o un tipo de datos
- field: nombre de la columna si difiere de la propiedad
- defaultValue: valor predeterminado
- allowNull: restricción NOT NULL
- unique: restricción UNIQUE
- primaryKey: clave principal, restricción NOT NULL y UNIQUE
- autoIncrement: Solo lectura, incremento automático
- validate: objeto de validaciones a ejecutar para esta columna cada vez que se guarda el modelo.
- onUpdate: CASCADE, RESTRICT, SET DEFAULT, SET NULL o NO ACTION
- onDelete: CASCADE, RESTRICT, SET DEFAULT, SET NULL o NO ACTION

Data Types

- const { DataTypes } = require("sequelize");
- **Strings**
- DataTypes.STRING // VARCHAR(255)
- DataTypes.STRING(1234) // VARCHAR(1234)
- DataTypes.STRING.BINARY // VARCHAR BINARY
- DataTypes.TEXT // TEXT
- DataTypes.TEXT('tiny') // TINYTEXT
- DataTypes.CITEXT // CITEXT PostgreSQL and SQLite only.
- DataTypes.TSVECTOR // TSVECTOR PostgreSQL only.
- **Boolean**
- DataTypes.BOOLEAN // TINYINT(1)
- **Numbers**
- DataTypes.INTEGER // INTEGER
- DataTypes.BIGINT // BIGINT
- DataTypes.BIGINT(11) // BIGINT(11)
- DataTypes.FLOAT // FLOAT
- DataTypes.FLOAT(11) // FLOAT(11)
- DataTypes.FLOAT(11, 10) // FLOAT(11,10)
- DataTypes.DOUBLE // DOUBLE
- DataTypes.DOUBLE(11) // DOUBLE(11)
- DataTypes.DOUBLE(11, 10) // DOUBLE(11,10)
- DataTypes.DECIMAL // DECIMAL
- DataTypes.DECIMAL(10, 2) // DECIMAL(10,2)
- *PostgreSQL only.*
- DataTypes.REAL // REAL
- DataTypes.REAL(11) // REAL(11) DataTypes.REAL(11, 12) // REAL(11,12
- **Dates**
- DataTypes.DATE // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE for postgres
- DataTypes.DATE(6) // DATETIME(6)
- DataTypes.DATEONLY // DATE without time
- **UUIDs**
- DataTypes.UUID
- **Other**
- DataTypes.RANGE
- DataTypes.ARRAY
- DataTypes.BLOB
- DataTypes.ENUM

Opciones de la tabla

- De manera predeterminada, cuando no se proporciona el nombre de la tabla, Sequelize pluraliza automáticamente el nombre del modelo y lo usa como nombre de la tabla.
- Para hacer que el nombre de la tabla sea igual al del modelo:
  { freezeTableName: true }
- Para proporcionar el nombre de la tabla directamente:
  { tableName: 'Employees' }
- De forma predeterminada, Sequelize agrega automáticamente los campos createdAt y updatedAt a cada modelo, utilizando el tipo de datos DataTypes.DATE.
- El campo createdAt contendrá una marca temporal que representa el momento de la creación.
- El campo updatedAt contendrá una marca temporal que representa el momento de la última actualización.
- Estos campos se administran automáticamente siempre que se use Sequelize para crear o actualizar. Este comportamiento se puede desactivar para un modelo con la opción:
  { timestamps: false }

Asociaciones

- Sequelize admite las asociaciones estándar: uno a uno, uno a muchos y muchos a muchos.
- Para ello, Sequelize proporciona cuatro tipos de asociaciones que deben combinarse para crearlas:
  - A.hasOne(B): existe una relación uno a uno entre A y B, con la clave ajena definida en el modelo de destino (B).
  - A.belongsTo(B): existe una relación uno a uno entre A y B, con la clave ajena definida en el modelo fuente (A).
  - A.hasMany(B): existe una relación de uno a muchos entre A y B, con la clave ajena definida en el modelo de destino (B).
  - La A.belongsToMany(B, { through: 'C' }): existe una relación muchos a muchos entre A y B, utilizando table C como tabla renacida, que tendrá las claves ajenas de A y B. Sequelize creará automáticamente este modelo C (a menos que ya exista) y definirá las claves ajenas apropiadas en él.

Asociaciones

- Estas llamadas harán que Sequelize agregue automáticamente claves ajenas a los modelos apropiados (a menos que ya estén presentes). Se pueden establecer manualmente con:

```
User.hasOne(Role, {
      foreignKey: {
        name: 'idUser'
      }
});
```

- Se puede automatizar la integridad referencia, acepta RESTRICT, CASCADE, NO ACTION, SET DEFAULT y SET NULL

```
User.hasOne(Role, {
  onUpdate: 'CASCADE ',
  onDelete: 'RESTRICT'
});
```


Sequelize-Auto (MySQL)

https://github.com/sequelize/sequelize-auto

- Genera automáticamente modelos para SequelizeJS a través de la línea de comandos mediante ingeniería inversa. Para instalarlo:
  - npm install sequelize mysql2
  - npm install sequelize-auto --save-dev
- Para generar los modelos con notación PascalCase (modelos), camelCase (propiedades) y kebab-case (ficheros):
  - npx sequelize-auto -o "./models" --cm p --cp c --cf k -e mysql -h localhost -p 3306 -d cursos -u root -x root
- Para acceder a los datos:

```
const { Sequelize, DataTypes } = require('sequelize');
const initModels = require("./models/init-models");
const sequelize = new Sequelize('mysql://root:root@localhost:3306/sakila')
const dbContext = initModels(sequelize);
dbContext.actor.findOne({where: { actor_id: 2 }}).then(row => console.log(row.toJSON()))
```

Sequelize-Auto (SQL Server)

https://github.com/sequelize/sequelize-auto

- Genera automáticamente modelos para SequelizeJS a través de la línea de comandos mediante ingeniería inversa. Para instalarlo:
  - npm install sequelize tedious
  - npm install sequelize-auto --save-dev
- Para generar los modelos con notación PascalCase (modelos), camelCase (propiedades) y kebab-case (ficheros):
  - npx sequelize-auto -o "./models" --cm p --cp c --cf k -e mssql -h localhost -p 1433 -d cursos -u sa -x P@$$w0rd
- Para acceder a los datos:

```
const { Sequelize, DataTypes } = require('sequelize');
const initModels = require("./models/init-models");
const sequelize = new Sequelize('cursos', 'sa', 'P@$$w0rd',
      { dialect: 'mssql', host: 'localhost', port: 1433, });
const dbContext = initModels(sequelize);
dbContext.actor.findOne({where: { actor_id: 2 }}).then(row => console.log(row.toJSON()))
```


db-context

- db-context.js

```
const { Sequelize } = require('sequelize');
const initModels = require("./init-models");
const sequelize = new Sequelize('cursos', 'sa', 'P@$$w0rd', {
   dialect: 'mssql',
   host: 'localhost',
   port: 1433,
 });
exports.sequelize = sequelize
exports.dbContext = initModels(sequelize)
```

- Importar:

```
const { sequelize, dbContext } = require('./model/db-context');
```

Consultas

- Para leer toda la tabla de la base de datos:
  const users = await User.findAll();
- Para seleccionar solo algunos atributos:
  const users = await User.findAll({ attributes: ['id', 'firstName'] });
- Los atributos se pueden renombrar usando una matriz anidada:
  { attributes: ['id', ['firstName', 'nombre']] }
- Se puede usar sequelize.fn para hacer agregaciones:
  { attributes: ['id', [sequelize.fn('COUNT', sequelize.col('likes')), 'n_likes'] }
- Del mismo modo, también es posible eliminar algunos atributos:
  { attributes: { exclude: ['lastName'] } }

Filtrado de Consultas

- La opción where se utiliza para filtrar la consulta. Hay muchos operadores para usar para la cláusula where, disponibles como Símbolos de Op.
  const users = await User.findAll({ where: { id: 2, status: 'active' } });
  const users = await User.findAll({ where: { id: { [Op.eq]: 2 } } });
- Para consultas de resultado único por la clave primaria:
  - const usr = await User.findByPk(2);
- Para obtener un resultado único o la primera entrada que encuentre (que cumple con las opciones de consulta opcionales, si se proporcionan).
  const usr = await User.findOne({ where: { id: 2} });
- Los símbolos Op:, son los operadores de filtrado de consultas:
  - [Op.and]:[] [Op.or]:[] [Op.not]:[]
  - [Op.eq][Op.ne][Op.gt][Op.gte][Op.lt][Op.lte][Op.is][Op.not]
  - [Op.in][Op.notIn][Op.between][Op.notBetween]
  - [Op.like][Op.notLike][Op.startsWith][Op.endsWith][Op.substring]
  - [Op.regexp][Op.notRegexp]
  - [Op.all][Op.any]

Asociaciones

- Los conceptos de Eager Loading y Lazy Loading son fundamentales para comprender cómo funcionan las asociaciones de búsqueda en Sequelize. Lazy Loading se refiere a la técnica de obtener los datos asociados solo cuando realmente son necesarios; Eager Loading, por otro lado, se refiere a la técnica de obtener todo a la vez, desde el principio, con una consulta más grande.
- Carga diferida:

```
const usr = await User.findOne({ where: { id: 2} });
const roles = await usr.getRoles()
```

- Carga ansiosa:

```
const usr = await User.findOne({
  where: { id: 2},
  include: { model: Rol, as: 'Roles' }
});
```


Ordenar y agrupar

- Sequelize proporciona las opciones order y group para trabajar con ORDER BY y GROUP BY.
- La opción order toma una matriz de elementos para ordenar la consulta o un método de secuencia. Estos elementos son en sí mismos matrices en la forma [column, direction]. La dirección se verificará en una lista blanca de direcciones válidas (como ASC, DESC, NULLS FIRST, etc.).

```
{ order: [['firstName', 'lastName', 'DESC'], sequelize.fn('max', sequelize.col('age'))] }
```

- Las opciones offset y limit permiten trabajar con paginación:

```
const users = await User.findAll({ offset: 50, limit: 10 });
```

- Sequelize también proporciona los métodos agregados count, max, min y sum:

```
const count = await User.count({ where: { id: { [Op.gt]: 25 }}});
const max = await User.max('age', { where: { age: { [Op.lt]: 20 } } });
```

Instancias de modelo

- Las instancias del modelo representan las filas de la tabla y cuentan con los métodos para hacer la persistencia.
- Para crear una nueva instancia:

```
const usr = User.build({ firstName: "Pepe" });
```

- Para guardar la instancia (INSERT o UPDATE):

```
await usr.save();
```

- Para crear una nueva instancia y guardarla (INSERT):

```
const usr = await User.create({ firstName: "Pepito" });
```

- Si se cambia el valor de algún campo de una instancia, al llamar a save() se actualizará en consecuencia (UPDATE):

```
usr.firstName = "John"
```

- Se puede actualizar varios campos a la vez con el método set:

```
usr.set({ firstName: "Pepito", lastName: "Grillo" });
```


Instancias de modelo

- El método save está optimizado internamente para actualizar solo los campos que realmente cambiaron. Esto significa que si no cambia nada y se llama a save, Sequelize sabrá que guardar es superfluo y no hará nada, es decir, no se generará ninguna consulta (devolverá una Promesa que se resolverá de inmediato).
- Para aumentar/disminuir los valores de una instancia sin tener problemas de simultaneidad, Sequelize proporciona los métodos de instancia .increment y .decrement

```
const incrementResult = await usr.increment('likes', { by: 1 });
```

- Se puede recargar una instancia desde la base de datos descartando las modificaciones:

```
await usr.reload();
```

- Para eliminar una instancia (DELETE):

```
await usr.destroy();
```

Validaciones y Restricciones

- Las validaciones son comprobaciones realizadas en el nivel Sequelize, en JavaScript puro. Pueden ser arbitrariamente complejas si se proporciona una función de validación personalizada, o pueden ser uno de los validadores integrados que ofrece Sequelize. Si falla una validación, no se enviará ninguna consulta SQL a la base de datos.
- Por otro lado, las restricciones son reglas definidas a nivel de SQL. Si falla una verificación de restricción, la base de datos arrojará un error y Sequelize reenviará este error a JavaScript, en este caso se realiza la consulta SQL, a diferencia del caso de las validaciones.

Validaciones

- Se pueden definir validadores personalizados o usar los validadores integrados, implementados por validator.js.
- Las validaciones se pueden establecer a nivel de atributo individual con la propiedad validate:

```
firstName: {
    type: DataTypes.STRING,
    allowNull: false,
    validate: {
        len: [2,10],
        esMayusculas(valor) {
            if(valor.toString().toUpperCase() !== valor)
                throw new Error('Debe estar en mayúsculas');
        }
    }
},
```

Validaciones

- Para usar un mensaje de error personalizado en lugar del proporcionado por validator.js:

```
validate: {
  is:{
    args: /^(\d{3}\s){2}\d{3}$/,
    msg: 'El formato debe ser: 555 999 999'
  }
 }
```

- Para hacer validaciones conjuntas se definen en la propiedad validate de las opciones del modelo:

```
validate: {
    bothCoordsOrNone() {
      if ((this.latitude === null) !== (this.longitude === null)) {
        throw new Error('Either both latitude and longitude, or neither!');
      }
    }
  }
```


Validaciones

```
is: /^[a-z]+$/i,                  // matches this RegExp
not: /^[a-z]+$/i,                 // does not match this RegExp
isEmail: true,                    // checks for email format (foo@bar.com)
isUrl: true,                      // checks for url format (https://foo.com)
isIP: true,                       // checks for IPv4 or IPv6 format
isIPv4: true,                     // checks for IPv4
isIPv6: true,                     // checks for IPv6 format
isAlpha: true,                    // will only allow letters
isAlphanumeric: true,             // will only allow alphanumeric characters
isNumeric: true,                  // will only allow numbers
isInt: true,                      // checks for valid integers
isFloat: true,                    // checks for valid floating point numbers
isDecimal: true,                  // checks for any numbers
isLowercase: true,                // checks for lowercase
isUppercase: true,                // checks for uppercase
notNull: true,                    // won't allow null
isNull: true,                     // only allows null
notEmpty: true,                   // don't allow empty strings
equals: 'specific value',         // only allow a specific value
contains: 'foo',                  // force specific substrings
notIn: [['foo', 'bar']],          // check the value is not one of these
isIn: [['foo', 'bar']],           // check the value is one of these
notContains: 'bar',               // don't allow specific substrings
len: [2,10],                      // only allow values with length between 2 and 10
isUUID: 4,                        // only allow uuids
isDate: true,                     // only allow date strings
isAfter: "2011-11-05",            // only allow date strings after a specific date
isBefore: "2011-11-05",           // only allow date strings before a specific date
max: 23,                          // only allow values <= 23
min: 23,                          // only allow values >= 23
isCreditCard: true,               // check for valid credit card numbers
```

Modificaciones por lotes

- Las consultas de actualización también aceptan la opción where, al igual que las consultas de lectura, para actualizar múltiples filas:

```
await User.update({ status: 'active' }, { where: { lastName: null } });
```

- Para eliminar múltiples filas:

```
await User.destroy({ where: { lastName: null } });
```

- Para eliminar todas las filas:

```
await User.destroy({truncate: true});
```

- Para insertar múltiples filas:

```
const users = await User.bulkCreate([
  { firstName: 'Jack', lastName: 'Sparrow' },
  { firstName: 'Davy', lastName: 'Jones' }
]);
```


Asociaciones

- El método save() de instancia no reconoce las asociaciones. En otras palabras, si se cambia un valor de un objeto secundario que se cargó con Eager junto con un objeto principal, llamar al save() principal ignorará por completo el cambio que ocurrió en el secundario.
- Para crear, actualizar y eliminar se puede:
  - Usar las consultas del modelo estándar directamente
  - Usar los métodos/mixins especiales disponibles para los modelos asociados
- Cuando se define una asociación entre dos modelos, las instancias de esos modelos obtienen métodos especiales para interactuar con sus contrapartes asociadas. Los nombres que Sequelize da a estos métodos especiales están formados por un prefijo concatenado con el nombre del modelo (con la primera letra en mayúscula) en singular (add, create, has, remove) para uno o en plural (add, count, get, has, remove, set) para muchos (array)

Asociaciones

```
const foo = await Foo.create({ name: 'the-foo' });
const bar1 = await Bar.create({ name: 'some-bar' });
const bar2 = await Bar.create({ name: 'another-bar' });
console.log(await foo.getBars()); // []
console.log(await foo.countBars()); // 0
console.log(await foo.hasBar(bar1)); // false
await foo.addBars([bar1, bar2]);
console.log(await foo.countBars()); // 2
await foo.addBar(bar1);
console.log(await foo.countBars()); // 2
console.log(await foo.hasBar(bar1)); // true
await foo.removeBar(bar2);
console.log(await foo.countBars()); // 1
await foo.createBar({ name: 'yet-another-bar' });
console.log(await foo.countBars()); // 2
await foo.setBars([]); // Un-associate all previously associated bars
```


Transacciones

- Sequelize no utiliza transacciones de forma predeterminada. Sin embargo, para su uso en producción, se debe configurar Sequelize para usar transacciones.
- Sequelize admite dos formas de utilizar transacciones:
  - Transacciones no administradas (manuales): el usuario debe realizar manualmente la confirmación y reversión de la transacción (llamando a los métodos Sequelize apropiados).
  - Transacciones administradas (automáticas): Sequelize revertirá automáticamente la transacción si se produce algún error o, en caso contrario, confirmará la transacción. Además, si CLS (Almacenamiento local de continuación) está habilitado, todas las consultas dentro de la devolución de llamada de la transacción recibirán automáticamente el objeto de la transacción.
- A través de las opciones se puede establecer el grado de aislamiento:

```
const { Transaction } = require('sequelize');
await sequelize.transaction({ isolationLevel: Transaction.ISOLATION_LEVELS.SERIALIZABLE }
```

Transacciones no administradas

- Se obtiene el objeto transaction, que abre la transacción, y con sus métodos se debe confirmar (.commit()) o descartar (.rollback()) explícitamente.

```
const t = await sequelize.transaction();
try {
  const user = await User.create({ firstName: 'Bart', lastName: 'Simpson'},
        { transaction: t });
  await user.addSibling({ firstName: 'Lisa', lastName: 'Simpson'},
        { transaction: t });
  await t.commit();
} catch (error) {
  await t.rollback();
}
```


Transacciones administradas

- Las transacciones administradas manejan la confirmación o reversión de la transacción automáticamente a través de una función, revertirá automáticamente la transacción si se produce algún error o, en caso contrario, confirmará la transacción.

```
const result = await sequelize.transaction(async (t) => {
  const user = await User.create({ firstName: 'Abraham', lastName: 'Lincoln' },
        { transaction: t });
  await user.setShooter({ firstName: 'John', lastName: 'Boothe' },
        { transaction: t });
  return user;
});
```

Modo Paranoico

- Sequelize admite el concepto de tablas paranoicas. Una tabla paranoica es aquella que, cuando se le dice que elimine un registro, no lo eliminará realmente. En su lugar, una columna especial llamada deletedAt contendrá una marca temporal de la solicitud de eliminación o null si no se ha borrado.
- Esto significa que las tablas paranoicas realizan una eliminación temporal de registros, en lugar de una eliminación permanente.
- Para convertir un modelo en paranoico, se debe pasar la opción paranoid: true a la definición del modelo (no funcionará si también aprueba timestamps: false). Se puede cambiar el nombre de la columna predeterminada (que es deletedAt) por otro.

```
{
    timestamps: false, // createdAt: false, updatedAt: false,
    paranoid: true,
    deletedAt: 'active'
}
```


Modo Paranoico

- Cuando llame al método destroy, se producirá una eliminación temporal (UPDATE asignando el timestamp a deletedAt).
- Para forzar una eliminación permanente en un modelo paranoico:

```
await user.destroy({ force: true });
await User.destroy({ where: { id: 1 }, force: true });
```

- Para restaurar registros eliminados temporalmente (UPDATE asignando el null a deletedAt), se puede usar el método restore:

```
await user.restore(); // si se conserva la instancia del modelo
await User.restore({ where: { id: 1 } });
```

- Cada consulta realizada por Sequelize ignorará automáticamente los registros eliminados temporalmente (excepto las consultas directas sin procesar).
- Si se desea permitir que la consulta vea los registros eliminados temporalmente, se puede pasar la opción paranoid: false al método de consulta.

```
await User.findAll({ where: { id: { [Op.lt]: 100 } }, paranoid: false });
```

Consultas sin procesar

- Como a menudo hay casos de uso en los que es más fácil ejecutar consultas SQL sin procesar o ya preparadas, puede usar el método sequelize.query.
- De forma predeterminada, la función devolverá dos argumentos: una matriz de resultados y un objeto que contiene metadatos (como la cantidad de filas afectadas, etc.). Dado que se trata de una consulta sin formato, los metadatos son específicos del dialecto.

```
const [results, metadata] = await sequelize.query("SELECT * FROM `contactos`");
```

- En los casos en los que no se necesite acceder a los metadatos, se puede pasar un tipo de consulta para decirle a Sequelize cómo formatear los resultados.

```
const users = await sequelize.query("SELECT * FROM `contactos`", { type: QueryTypes.SELECT });
```

- Si se pasa un modelo, los datos devueltos serán instancias de ese modelo.

```
const users = await sequelize.query("SELECT * FROM `contactos`", { model: User, mapToModel: true });
```


Consultas parametrizadas

- Las consultas parametrizadas se pueden definir como reemplazos o parámetros de enlace. Los reemplazos se escapan y se insertan en la consulta mediante la secuenciación antes de que la consulta se envíe a la base de datos, mientras que los parámetros de vinculación se envían a la base de datos fuera del texto de la consulta SQL.
- Los reemplazos en una consulta se pueden realizar de dos maneras diferentes, ya sea utilizando parámetros con nombre (que comienzan con :) o con marcadores de posición, representados por un ?. Los reemplazos se pasan en la propiedad replacements de opciones.
  - Si se pasa una matriz, ? se reemplazará en el orden en que aparecen en la matriz

    ```
    const users = await sequelize.query('SELECT * FROM `contactos` where idContacto=?', { replacements: [1], type: QueryTypes.SELECT });
    ```

  - Si se pasa un objeto, :nombre se reemplazará con las claves de ese objeto. Si el objeto contiene claves que no se encuentran en la consulta o viceversa, se generará una excepción.

    ```
    const users = await sequelize.query('SELECT * FROM `contactos` where idContacto=:id', { replacements: {id:1}, type: QueryTypes.SELECT });
    ```

Consultas parametrizadas

- Se hace referencia a los parámetros de vinculación mediante $1, $2, ... (numérico) o $nombre (alfanumérico). Esto es independiente del dialecto. Con $$ se puede utilizar para escapar de un signo literal $. Los parámetros de vinculación se pasan en la propiedad bind de opciones.
  - Si se pasa una matriz, $1 se vincula al primer elemento de la matriz (bind[0])

    ```
    const users = await sequelize.query('SELECT * FROM `contactos` where idContacto=$1', { bind: [1], type: QueryTypes.SELECT });
    ```
  - Si se pasa un objeto, $nombre está vinculado a object['key']. Cada clave debe comenzar con un carácter no numérico. La base de datos puede agregar más restricciones a esto: Los parámetros de enlace no pueden ser palabras clave de SQL, ni nombres de tablas o columnas, también se ignoran en el texto o los datos citados..

    ```
    const users = await sequelize.query('SELECT * FROM `contactos` where idContacto=$id', { bind: {id:1}, type: QueryTypes.SELECT });
    ```
- La matriz u objeto en replacements o bind debe contener todos los valores enlazados/reemplazados o Sequelize generará una excepción, incluso si la base de datos puede ignorar el parámetro vinculado.

RepresentationalState Transfer

SERVICIOS RESTFUL

REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba.
- El HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado:
  - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- El HTTP permite en el encabezado transmitir la información de comportamiento:
  - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, …

Objetivos de los servicios REST

- Desacoplar el cliente del backend
- Mayor escalabilidad
  - Sin estado en el backend.
- Separación de problemas
- División de responsabilidades
- API uniforme para todos los clientes
  - Disponer de una interfaz uniforme (basada en URIs)

Estilo de arquitectura

- Las APIs de REST se diseñan en torno a recursos, que son cualquier tipo de objeto, dato o servicio al que puede acceder el cliente.
- Un recurso tiene un identificador, que es un URI que identifica de forma única ese recurso.
- Los clientes interactúan con un servicio mediante el intercambio de representaciones de recursos, negociados a través de formatos MIME y las cabeceras Accept y Content-type.
- Las APIs de REST usan una interfaz uniforme, que ayuda a desacoplar las implementaciones de clientes y servicios. En las APIs REST basadas en HTTP, la interfaz uniforme incluye el uso de verbos HTTP estándar para realizar operaciones en los recursos. Las operaciones más comunes son GET, POST, PUT, PATCH y DELETE. El código de estado de la respuesta indica el éxito o error de la petición.
- Las APIs de REST usan un modelo de solicitud sin estado. Las solicitudes HTTP deben ser independientes y pueden producirse en cualquier orden, por lo que no es factible conservar la información de estado transitoria entre solicitudes. El único lugar donde se almacena la información es en los propios recursos y cada solicitud debe ser una operación atómica.
- Las APIs de REST se controlan mediante vínculos de hipermedia.

Estilo de arquitectura

- Métodos GET
  - Petición de consulta a la URL sin identificador (todas las instancias del recurso) o con identificador (una instancia) y la cabecera accept para la respuesta.
  - Normalmente, un método GET correcto devuelve el código de estado HTTP 200 (Correcto), el cuerpo de respuesta contiene una representación del recurso y la cabecera content-type. Si no se encuentra el recurso, el método debe devolver HTTP 404 (No encontrado).
- Métodos POST
  - Petición de crear o reemplazar a la URL sin identificador, la instancia en el cuerpo, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Si un método POST crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), el URI del nuevo recurso se incluye en el encabezado Location de la respuesta. Con el código de estado HTTP 200, el cuerpo de respuesta contiene una representación del recurso y la cabecera content-type.
  - Si no crea un nuevo recurso, puede devolver el código de estado HTTP 200 e incluir el resultado de la operación en el cuerpo de respuesta y la cabecera content-type. O bien, si no hay ningún resultado para devolver, devolverá el código de estado HTTP 204 (Sin contenido) y sin cuerpo de respuesta.
  - Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver el código de estado HTTP 400 (Solicitud incorrecta), un 409 (Conflicto) o un 412 (fallo de concurrencia). El cuerpo de respuesta puede contener información adicional sobre el error o un vínculo a un URI que proporciona más detalles.

Estilo de arquitectura

- Métodos PUT
  - Petición de crear o reemplazar (idempotente) a la URL con identificador, la instancia en el cuerpo, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Al igual que con un método POST, si un método PUT crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), y si actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto), un 412 (fallo de concurrencia) o el recurso no existe, puede devolver un 404 (No encontrado).
  - Hay que considerar la posibilidad de implementar operaciones HTTP PUT masivas que pueden procesar por lotes las actualizaciones de varios recursos de una colección. La solicitud PUT debe especificar el URI de la colección y el cuerpo de solicitud debe especificar los detalles de los recursos que se van a modificar. Este enfoque puede ayudar a reducir el intercambio de mensajes y mejorar el rendimiento.
- Métodos DELETE
  - Petición de borrado a la URL sin identificador (todas las instancias del recurso) o con identificador (una instancia)
  - El método debe responder con un 204 (Sin contenido), que indica que la operación de eliminación es correcta, el cuerpo de respuesta no contiene información adicional. Si el recurso no existe, puede devolver un 404 (No encontrado), un 409 (Conflicto) o un 412 (fallo de concurrencia).

Estilo de arquitectura

- Métodos PATCH
  - Petición de reemplazo parcial a la URL con identificador, en el cuerpo el conjunto de cambios a aplicar, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Con una solicitud PATCH, el cliente envía un conjunto de actualizaciones a un recurso existente, en forma de un documento de revisión. El servidor procesa el documento de revisión para realizar la actualización que, por definición, no es idempotente..
  - La estructura del documento de revisión debería de seguir una sintaxis estándar como JSON Patch (RFC6902) o JSON Merge (RFC7386).
  - Los formatos MIME del documento de revisión aceptados por el servidor deberían aparecer en la cabecera Accept-Patch en respuesta a una petición OPTIONS.
  - Si el método actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud o el documento de revisión tiene un formato incorrecto, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) o un 412 (fallo de concurrencia), si no se admite el formato de documento de revisión devolverá un 415 (Tipo de medio no compatible) o el recurso no existe, puede devolver un 404 (No encontrado).

Estilo de arquitectura

- Request: Método /uri?parámetros
  - GET: Recupera el recurso (200)
    - Todos: Sin identificador
    - Uno: Con identificador
  - POST: Crea o reemplaza un nuevo recurso (201)
  - PUT: Crea o reemplaza el recurso identificado (200, 204)
  - DELETE: Elimina el recurso (204)
    - Todos: Sin identificador
    - Uno: Con identificador
- Cabeceras:
  - Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
  - Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- HTTP Status Code: Código de estado con el que el servidor informa del resultado de la petición.

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- Nivel 0: Definir un URI y todas las operaciones son solicitudes POST a este URI.
- Nivel 1 (Pobre): Crear distintos URI para recursos individuales pero utilizan solo un método.
  - Se debe identificar un recurso
    /entities/?invoices =2 → entities/invoices/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- Nivel 2 (Medio): Usar métodos HTTP para definir operaciones en los recursos.
- Nivel 3 (Óptimo): Usar hipermedia (HATEOAS, se describe a continuación).

Hypermedia

- Uno de los principales propósitos que se esconden detrás de REST es que debe ser posible navegar por todo el conjunto de recursos sin necesidad de conocer el esquema de URI. Cada solicitud HTTP GET debe devolver la información necesaria para encontrar los recursos relacionados directamente con el objeto solicitado mediante los hipervínculos que se incluyen en la respuesta, y también se le debe proporcionar información que describa las operaciones disponibles en cada uno de estos recursos.
- Este principio se conoce como HATEOAS, del inglés Hypertext as the Engine of Application State (Hipertexto como motor del estado de la aplicación). El sistema es realmente una máquina de estado finito, y la respuesta a cada solicitud contiene la información necesaria para pasar de un estado a otro; ninguna otra información debería ser necesaria.
- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- HATEOAS: Content-Type: application/hal+json

```
{
   "_links": {
     "self": {"href": "/orders/523" },
     "warehouse": {"href": "/warehouse/56" },
     "invoice": {"href": "/invoices/873"}
   },
   , "currency": "USD"
   , "status": "shipped"
   , "total": 10.20
}
```

Diseño de un Servicio Web REST

- Para el desarrollo de los Servicios Web's REST es necesario definir una serie de cosas:
  - Analizar el/los recurso/s a implementar
  - Diseñar la REPRESENTACION del recurso.
    - Deberemos definir el formato de trabajo del recurso: XML, JSON, HTML, imagen, RSS, etc
  - Definir el URI de acceso.
    - Deberemos indicar el/los URI de acceso para el recurso
  - Establecer los métodos soportados por el servicio
    - GET, POST, PUT, DELETE
  - Fijar qué códigos de estado pueden ser devueltos
    - Los códigos de estado HTTP típicos que podrían ser devueltos
- Todo lo anterior dependerá del servicio a implementar.

Definir operaciones

- Sumario y descripción de la operación.
- Dirección: URL
  - Sin identificador
  - Con identificador
  - Con parámetros de consulta
- Método: GET | POST | PUT | DELETE | PATCH
- Solicitud:
  - Cabeceras:
    - ACCEPT: formatos aceptables si espera recibir datos
    - CONTENT-TYPE: formato de envío de los datos en la solicitud
    - Otras cabeceras: Authorization, Cache-control, X-XSRF-TOKEN, …
  - Cuerpo: en caso de envío, estructura de datos formateados según el CONTENT-TYPE.
- Respuesta:
  - Cabeceras:
    - Códigos de estado HTTP: posibles y sus causas.
    - CONTENT-TYPE: formato de envío de los datos en la respuesta
    - Otras cabeceras
  - Cuerpo: en caso de respuesta, estructura de datos según código de estado y formateados según el CONTENT-TYPE.

Características de una API bien diseñada

- **Fácil de leer y trabajar**: con una API bien diseñada será fácil trabajar, y sus recursos y operaciones asociadas pueden ser memorizados rápidamente por los desarrolladores que trabajan con ella constantemente.
- **Difícil de usar mal**: la implementación e integración con una API con un buen diseño será un proceso sencillo, y escribir código incorrecto será un resultado menos probable porque tiene comentarios informativos y no aplica pautas estrictas al consumidor final de la API.
- **Completa y concisa**: Finalmente, una API completa hará posible que los desarrolladores creen aplicaciones completas con los datos que expone. Por lo general, la completitud ocurre con el tiempo, y la mayoría de los diseñadores y desarrolladores de API construyen gradualmente sobre las APIs existentes. Es un ideal por el que todo ingeniero o empresa con una API debe esforzarse.

Guía de diseño

- Organización de la API en torno a los recursos
- Definición de operaciones en términos de métodos HTTP
- Conformidad con la semántica HTTP
- Filtrado y paginación de los datos
- Compatibilidad con respuestas parciales en recursos binarios de gran tamaño
- Uso de HATEOAS para permitir la navegación a los recursos relacionados
- Control de versiones en la API RESTful
- Documentación Open API

Definición de recursos

- La organización de la API en torno a los recursos se centran en las entidades de dominio que debe exponer la API. Por ejemplo, en un sistema de comercio electrónico, las entidades principales podrían ser clientes y pedidos. La creación de un pedido se puede lograr mediante el envío de una solicitud HTTP POST que contiene la información del pedido. La respuesta HTTP indica si el pedido se realizó correctamente o no.
- Un recurso no tiene que estar basado en un solo elemento de datos físico o tablas de una base de datos relacional. La finalidad de REST es modelar entidades y las operaciones que un consumidor externo puede realizar sobre esas entidades, no debe exponerse a la implementación interna.
- Es necesario adoptar una convención de nomenclatura coherente para los URI. Los URI de recursos deben basarse en nombres (de recurso), nunca en verbos (las operaciones en el recurso) y, en general, resulta útil usar nombres plurales que hagan referencia a colecciones. Debe seguir una estructura jerárquica que refleje las relaciones entre los diferentes tipos de recursos.
- Hay que considerar el uso del enfoque HATEOAS para permitir el descubrimiento y la navegación a los recursos relacionados o el enfoque del patrón agregado.

Definición de recursos

- Exponer una colección de recursos con un único URI puede dar lugar a que las aplicaciones capturen grandes cantidades de datos cuando solo se requiere un subconjunto de la información. A través de la definición de parámetros de cadena de consulta se pueden realizar particiones horizontales con filtrado, ordenación y paginación, o particiones verticales con la proyección de las propiedades a recuperar:
  - https://host/users?page=1&rows=20&projection=userId,name,lastAccess
- La definición de operaciones con el recurso se realiza en términos de métodos HTTP, estableciendo cuales serán soportadas. Las operaciones no soportadas por métodos HTTP deben sustantivarse al crearles una URI especifica y utilizar el método HTTP semánticamente mas próximo.
  - DELETE https://host/users/bloqueo (desbloquear)
  - POST https://host/pedido/171/factura (facturar)
- Hay que establecer el tipo o tipos de formatos mas adecuados para las representaciones de recursos. Los formatos se especifican mediante el uso de tipos de medios, también denominados tipos MIME. En el caso de datos no binarios, la mayoría de las APIs web admiten JSON (application/json) y, posiblemente, XML (application/xml).

Políticas de versionado

- Es muy poco probable que una API permanezca estática. Conforme los requisitos empresariales cambian, se pueden agregar nuevas colecciones de recursos, las relaciones entre los recursos pueden cambiar y la estructura de los datos de los recursos debe adecuarse.
- Los cambios rupturistas no son compatibles con la versión anterior, el consumidor tendrá que adaptar su código para pasar su aplicación existente a la nueva versión y evitar que se rompa.
- Hay dos razones principales por las que las APIs HTTP se comportan de manera diferente al resto de las APIs:
  - El código del cliente dicta lo que lo romperá: Un proveedor de API no tiene control sobre las herramientas que un consumidor puede usar para interpretar una respuesta de la API y la tolerancia al cambio que tienen esas herramientas varían ampliamente, si es rupturista o no.
  - El proveedor de API elige si los cambios son opcionales o transparentes: Los proveedores de API pueden actualizar su API y los cambios en las respuestas afectarán inmediatamente a los clientes. Los clientes no pueden decidir si adoptar o no la nueva versión, lo que puede generar fallos en cascada en los cambios rupturistas.

Políticas de versionado

- El control de versiones permite que una API indique la versión expuesta y que una aplicación cliente pueda enviar solicitudes que se dirijan a una versión específica con una característica o un recurso.
  - Sin control de versiones: Este es el enfoque más sencillo y puede ser aceptable para algunas APIs internas. Los grandes cambios podrían representarse como nuevos recursos o nuevos vínculos.
  - Control de versiones en URI: Cada vez que modifica la API web o cambia el esquema de recursos, agrega un número de versión al URI para cada recurso. Los URI ya existentes deben seguir funcionando como antes y devolver los recursos conforme a su esquema original.

    http://host/v2/users
  - Control de versiones en cadena de consulta: En lugar de proporcionar varios URI, se puede especificar la versión del recurso mediante un parámetro dentro de la cadena de consulta anexada a la solicitud HTTP:

    http://host/users?versión=2.0

Políticas de versionado

- Control de versiones en encabezado: En lugar de anexar el número de versión como un parámetro de cadena de consulta, se podría implementar un encabezado personalizado que indica la versión del recurso. Este enfoque requiere que la aplicación cliente agregue el encabezado adecuado a las solicitudes, aunque el código que controla la solicitud de cliente puede usar un valor predeterminado (versión actual) si se omite el encabezado de versión.

  ```
  GET https://host/users HTTP/1.1
  Custom-Header: api-version=1
  ```

- Control de versiones por MIME (tipo de medio): Cuando una aplicación cliente envía una solicitud HTTP GET a un servidor web, debe prever el formato del contenido que puede controlar mediante el uso de un encabezado Accept.

  ```
  GET https:// host/users/3 HTTP/1.1
  Accept: application/vnd.mi-api.v1+json
  ```

- Si la versión no está soportada, el servicio podría generar un mensaje de respuesta HTTP 406 (no aceptable) o devolver un mensaje con un tipo de medio predeterminado.
- Los esquemas de control de versiones de URI y de cadena de consulta son compatibles con la caché HTTP puesto que la misma combinación de URI y cadena de consulta hace referencia siempre a los mismos datos.

Políticas de versionado

- Dentro de la política de versionado es conveniente planificar la obsolescencia y la política de desaprobación.
- La obsolescencia programada establece el periodo máximo, como una franja temporal o un número de versiones, en que se va a dar soporte a cada versión, evitando los sobrecostes derivados de mantener versiones obsoletas indefinidamente.
- Dentro de la política de desaprobación, para ayudar a garantizar que los consumidores tengan tiempo suficiente y una ruta clara de actualización, se debe establecer el número de versiones en que se mantendrá una característica marcada como obsoleta antes de su desaparición definitiva.
- La obsolescencia programada y la política de desaprobación beneficia a los consumidores de la API porque proporcionan estabilidad y sabrán qué esperar a medida que las APIs evolucionen.
- Para mejorar la calidad y avanzar las novedades, se podrán realizar lanzamientos de versiones Beta y Release Candidatos (RC) o revisiones para cada versión mayor y menor. Estas versiones provisionales desaparecerán con el lanzamiento de la versión definitiva.

Guía de implementación

- Procesamiento de solicitudes
  - Las acciones GET, PUT, DELETE, HEAD y PATCH deben ser idempotentes.
  - Las acciones POST que crean nuevos recursos no deben tener efectos secundarios no relacionados.
  - Evitar implementar operaciones POST, PUT y DELETE que generen mucha conversación.
  - Seguir la especificación HTTP al enviar una respuesta.
  - Admitir la negociación de contenido.
  - Proporcionar vínculos que permitan la navegación y la detección de recursos de estilo HATEOAS.

Guía de implementación

- Administración de respuestas y solicitudes de gran tamaño
  - Optimizar las solicitudes y respuestas que impliquen objetos grandes.
  - Admitir la paginación de las solicitudes que pueden devolver grandes cantidades de objetos.
  - Implementar respuestas parciales para los clientes que no admitan operaciones asincrónicas.
  - Evitar enviar mensajes de estado 100-Continuar innecesarios en las aplicaciones cliente.
- Mantenimiento de la capacidad de respuesta, la escalabilidad y la disponibilidad
  - Ofrecer compatibilidad asincrónica para las solicitudes de ejecución prolongada.
  - Comprobar que ninguna de las solicitudes tenga estado.
  - Realizar un seguimiento de los clientes e implementar limitaciones para reducir las posibilidades de ataques de denegación de servicio.
  - Administrar con cuidado las conexiones HTTP persistentes.

Guía de implementación

- Control de excepciones
  - Capturar todas las excepciones y devolver una respuesta significativa a los clientes.
  - Distinguir entre los errores del lado cliente y del lado servidor.
  - Evitar las vulnerabilidades por exceso de información.
  - Controlar las excepciones de una forma coherente y registrar la información sobre los errores.
- Optimización del acceso a los datos en el lado cliente
  - Admitir el almacenamiento en caché del lado cliente.
  - Proporcionar ETags para optimizar el procesamiento de las consultas.
  - Usar ETags para admitir la simultaneidad optimista.

Guía de implementación

- Publicación y administración de una API web
  - Todas las solicitudes deben autenticarse y autorizarse, y debe aplicarse el nivel de control de acceso adecuado.
  - Una API web comercial puede estar sujeta a diversas garantías de calidad relativas a los tiempos de respuesta. Es importante asegurarse de que ese entorno de host es escalable si la carga puede variar considerablemente con el tiempo.
  - Puede ser necesario realizar mediciones de las solicitudes para fines de monetización.
  - Es posible que sea necesario regular el flujo de tráfico a la API web e implementar la limitación para clientes concretos que hayan agotado sus cuotas.
  - Los requisitos normativos podrían requerir un registro y una auditoría de todas las solicitudes y respuestas.
  - Para garantizar la disponibilidad, puede ser necesario supervisar el estado del servidor que hospeda la API web y reiniciarlo si hiciera falta.

Guía de implementación

- Pruebas de la API
  - Ejercitar todas las rutas y parámetros para comprobar que invocan las operaciones correctas.
  - Verificar que cada operación devuelve los códigos de estado HTTP correctos para diferentes combinaciones de entradas.
  - Comprobar que todas las rutas estén protegidas correctamente y que estén sujetas a las comprobaciones de autenticación y autorización apropiadas.
  - Verificar el control de excepciones que realiza cada operación y que se devuelve una respuesta HTTP adecuada y significativa de vuelta a la aplicación cliente.
  - Comprobar que los mensajes de solicitud y respuesta están formados correctamente.
  - Comprobar que todos los vínculos dentro de los mensajes de respuesta no están rotos.

express.json()

- Es una función de middleware incorporada en Express. Analiza las solicitudes entrantes en formato JSON y se basa en el body-parser (analizador del cuerpo).
- Solo analiza JSON y solo mira las solicitudes donde el encabezado Content-Type coincide con la opción type (por defecto: "application/json"). Este analizador acepta cualquier codificación Unicode del cuerpo y admite la descompresión automática de codificaciones gzip y deflate.
- Se crea un nuevo objeto que contiene los datos analizados y se asigna a la propiedad body del objeto request después del middleware (es decir req.body) o un objeto vacío ({}) si no había cuerpo para analizar, el Content-Type no coincidía o se produjo un error.
- Como req.body se basa en la entrada enviada por el usuario, todas las propiedades y valores en este objeto no son confiables y deben validarse antes de utilizarse.

```
app.use(express.json())  // parse application/json
```

GET

- Recupera el recurso

```
app.get(servicio.url, async function (req, res) {
    // Obtener listado
    res.status(200).json(listado)
}
```

- Recupera el recurso identifcado

```
app.get(servicio.url  + '/:id', async function (req, res) {
    // Obtener elemento
    if (elemento) {
      res.status(200).json(elemento)
    } else {
      res.status(404).end()
    }
}
```


POST, PUT, DELETE

- Crear o reemplazar el recurso

```
app.post(servicio.url, async function (req, res) {
    // Create or replace
    res.status(OK ? 201 : 400).end()
}
```

- Crear o reemplazar el recurso identificado

```
app.put(servicio.url  + '/:id', async function (req, res) {
    // Create or replace
    if (elemento) {
        res.status(200).json(elemento)
    } else { res.status(400).end() }
}
```

- Añadir cabecera location a las respuestas 201:

```
res.append('location', `${req.protocol}://${req.hostname}:${req.connection.localPort}${req.originalUrl}/${newid}`)
```

- Borrar el recurso identificado

```
app.delete(servicio.url  + '/:id', async function (req, res) {
    // Delete
    res.status(OK ? 204 : 404).end()
}
```

Respuestas de error

- Un requisito común para los servicios REST es incluir detalles en el cuerpo de las respuestas de error. Se recomienda implementar la especificación RFC 7807 para "Detalles del problema para las API HTTP".
- El objeto de detalles del problema puede tener los siguientes miembros:
  - "type" (cadena): URI que identifica el tipo de problema y proporciona documentación legible por humanos para el tipo de problema. El valor "about:blank" (predeterminado) indica que el problema no tiene semántica adicional a la del código de estado HTTP.
  - "title" (cadena): Breve resumen legible por humanos del problema escribe. NO DEBE cambiar de una ocurrencia a otra del mismo problema, excepto para fines de localización. Con "type": "about:blank", DEBE coincidir con la versión textual del status.
  - "status" (número): Código de estado HTTP (por conveniencia, opcional, debe coincidir).
  - "detail" (cadena): Explicación legible por humanos específica de la ocurrencia concreta del problema.
  - "instance" (cadena): URI de referencia que identifica el origen de la ocurrencia del problema.
- Las definiciones de tipo de problema PUEDEN extender el objeto con miembros adicionales.

Restricciones de Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - **https**://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com**:8080**/scripts/codigo2.js
  - http://**scripts**.ejemplo.com/codigo2.js
  - http://**192.168.0.1**/scripts/codigo2.js
- La propiedad document.domain se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación.

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece y, por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.
- XMLHttpRequest sigue la política de mismo-origen, por lo que solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron a los proveedores de navegadores que permitieran a XMLHttpRequest realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE "
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

CORS middleware

- Implementación manual:

```
app.use(function (req, res, next) {
  var origen = req.header("Origin")
  if (!origen) origen = '*'
  res.header('Access-Control-Allow-Origin', origen)
  res.header('Access-Control-Allow-Headers', 'Origin, Content-Type, Accept, Authorization, X-
      Requested-With, X-XSRF-TOKEN')
  res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS')
  res.header('Access-Control-Allow-Credentials', 'true')
  next()
 })
```

- Paquete CORS de Connect/Express

```
$ npm install cors --save
var cors = require('cors')
app.use(cors())
```

JSON Schema

https://json-schema.org/

- JSON Schema es una especificación para definir, anotar y validar las estructuras de datos JSON. La especificación tiene varias definiciones formales y versiones.
- Un esquema de JSON contiene a qué versión de la especificación se ajusta, el identificador o la ubicación del esquema, un título, una descripción y el tipo de objeto del documento raíz.
- Define qué propiedades junto con sus tipos ha de contener el documento JSON al que se aplica, cuáles de esas propiedades son obligatorias y las validaciones sobre los datos como restricciones en los valores de los datos o elementos de un array.
- Un esquema permite el anidamiento de estructuras en las que también se definen que propiedades contienen y cuáles son requeridas.
- Un esquema JSON permite referenciar un esquema JSON externo.

Esquemas de datos

- Los tipos base son string, number, integer, boolean, array y object.
- Con la propiedad format se pueden especificar otros tipos especiales partiendo de los tipos base: long, float, double, byte, binary, date, dateTime, password.
- Los tipos array se definen como una colección de ítems y en dicha propiedad se define el tipo y la estructura de los elementos que lo componen. Los objetos son un conjunto de propiedades, cada una definida dentro de properties.
- Cada tipo y propiedad se identifica por un nombre que no debe estar repetido en su ámbito.
- Cada propiedad puede definir description, default, minimum, maximum, maxLength, minLength, pattern, required, readOnly, …
- Para una propiedad se pueden definir varios tipos (tipos mixtos o unión).
- Los tipos pueden hacer referencia a otros tipos.

Tipos de datos

type	format	Comentarios
boolean		Booleanos: true y false
integer	int32	Enteros con signo de 32 bits
integer	int64	Enteros con signo de 64 bits (también conocidos como largos)
number	float	Reales cortos
number	double	Reales largos
string		Cadenas de caracteres
string	password	Una pista a las IU para ocultar la entrada.
string	date	Según lo definido por full-date RFC3339 (2018-11-13)
string	date-time	Según lo definido por date-time- RFC3339 (2018-11-13T20:20:39+00:00)
string	byte	Binario codificados en base64
string	binary	Binario en cualquier secuencia de octetos
array		Colección de items
object		Colección de properties

Propiedades de los objetos de esquema

- type: integer, number, boolean, string, array, object
- format: long, float, double, byte, binary, date, dateTime, password
- title: Nombre a mostrar en el UI
- description: Descripción de su uso
- maximum: Valor máximo
- exclusiveMaximum: Valor menor que
- minimum: Valor mínimo
- exclusiveMinimum: Valor mayor que
- multipleOf: Valor múltiplo de
- maxLength: Longitud máxima
- minLength: Longitud mínima
- pattern: Expresión regular del patrón
- deprecated: Si está obsoleto y debería dejar de usarse
- nullable: Si acepta nulos
- default: Valor por defecto
- enum: Lista de valores con nombre
- example: Ejemplo de uso
- externalDocs: referencia a documentación externa adicional
- items: Definición de los elementos del array
- maxItems: Número máximo de elementos
- minItems: Número mínimo de elementos
- uniqueItems: Elementos únicos
- properties: Definición de las propiedades del objeto,
- maxProperties: Número máximo de propiedades
- minProperties: Número mínimo de propiedades
- readOnly: propiedad de solo lectura
- writeOnly: propiedad de solo escritura
- additionalProperties: permite referenciar propiedades adicionales
- required: Lista de propiedades obligatorias

schema

```
{
    "definitions": {},
    "$schema": "http://json-schema.org/draft-07/schema#",
    "$id": "https://example.com/personas.schema.json",
    "title": "Root",
    "type": "array",
    "default": [],
    "items":{
        "$id": "#root/items",
        "title": "Items",
        "type": "object",
        "required": [
            "id",
            "nombre"
        ],
        "properties": {
            "id": {
                "$id": "#root/items/id",
                "title": "Id",
                "type": "integer",
                "examples": [
                    2
                ],
                "default": 0
            },
            "nombre": {
                "$id": "#root/items/nombre",
                "title": "Nombre",
                "type": "string",
                "default": "",
                "examples": [
                    "Pepito"
                ],
                "pattern": "^.*$"
            },
            "apellidos": {
                "$id": "#root/items/apellidos",
                "title": "Apellidos",
                "type": "string",
                "default": "",
                "examples": [
                    "Grillo"
                ],
                "pattern": "^.*$"
            },
            "edad": {
                "$id": "#root/items/edad",
                "title": "Edad",
                "type": "integer",
                "examples": [
                    67
                ],
                "default": 0
            }
        }
    }
}
```


Validar un JSON

- Se puede usar una librería llamada ajv para la validación junto con otra llamada 'ajv-formats' para validar los campos format del schema.
  - npm install ajv ajv-formats
- Para validar el schema:

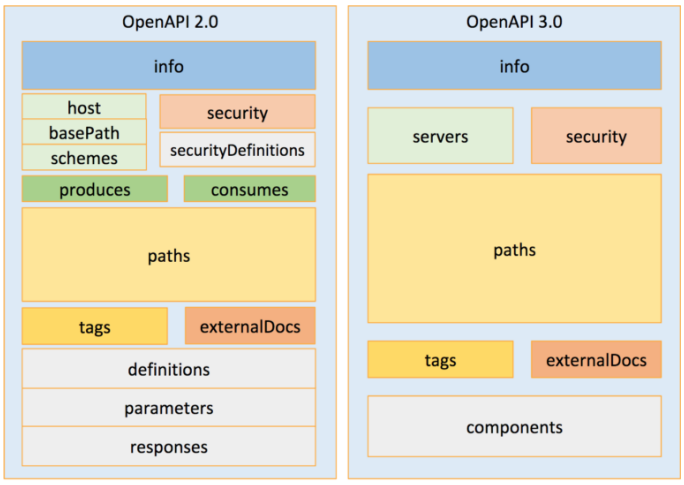
```
const Ajv = require("ajv")
const addFormats = require("ajv-formats")

const ajv = new Ajv()
addFormats(ajv)
const schema = { … }, data = { … }

const validate = ajv.compile(schema)
const valid = validate(data)
if (!valid) console.log(validate.errors)
```

OpenAPI

https://www.openapis.org/

- OpenAPI es un estándar para definir contratos de API. Los cuales describen la interfaz de una serie de servicios que vamos a poder consumir por medio de una signatura. Conocido previamente como Swagger, ha sido adoptado por la Linux Foundation y obtuvo el apoyo de compañías como Google, Microsoft, IBM, Paypal, etc. para convertirse en un estándar para las APIs REST.
- Las definiciones de OpenAPI se pueden escribir en JSON o YAML. La versión actual de la especificación es la 3.0.3 y orientada a YAML y la versión previa la 2.0, que es idéntica a la especificación 2.0 de Swagger antes de ser renombrada a “Open API Specification”.
- Actualmente nos encontramos en periodo de transición de la versión 2 a la 3, sin soporte en muchas herramientas.

Cambio de versión



Sintaxis

- Un documento de OpenAPI que se ajusta a la especificación de OpenAPI es en sí mismo un objeto JSON con propiedades, que puede representarse en formato JSON o YAML.
- YAML es un lenguaje de serialización de datos similar a XML pero que utiliza el sangrado para indicar el anidamiento, estableciendo la estructura jerárquica, y evitar la necesidad de tener que cerra los elementos.
- Para preservar la capacidad de ida y vuelta entre los formatos YAML y JSON, se RECOMIENDA la versión 1.2 de YAML junto con algunas restricciones adicionales:
  - Las etiquetas DEBEN limitarse a las permitidas por el conjunto de reglas del esquema JSON.
  - Las claves utilizadas en los mapas YAML DEBEN estar limitadas a una cadena escalar, según lo definido por el conjunto de reglas del esquema YAML Failsafe.
- Todos los nombres de propiedades o campos de la especificación distinguen entre mayúsculas y minúsculas. Esto incluye todas los propiedades que se utilizan como claves asociativas, excepto donde se indique explícitamente que las claves no distinguen entre mayúsculas y minúsculas.
- El esquema expone dos tipos de propiedades:
  - propiedades fijas: tienen el nombre establecido en el estándar
  - propiedades con patrón: sus nombres son de creación libre pero deben cumplir una expresión regular (patrón) definida en el estándar y deben ser únicos dentro del objeto contenedor.

Sintaxis

- El sangrado utiliza espacios en blanco, no se permite el uso de caracteres de tabulación.
- Los miembros de las listas van entre corchetes ([ ]) y separados por coma espacio (,), o uno por línea con un guion (-) inicial.
- Los vectores asociativos se representan usando los dos puntos seguidos por un espacio, "clave: valor", bien uno por línea o entre llaves ({ }) y separados por coma seguida de espacio (,).
- Un valor de un vector asociativo viene precedido por un signo de interrogación (?), lo que permite que se construyan claves complejas sin ambigüedad.
- Los valores sencillos (o escalares) por lo general aparecen sin entrecomillar, pero pueden incluirse entre comillas dobles ("), o apostrofes (').

Sintaxis

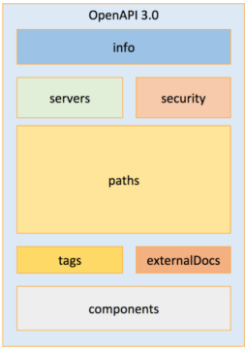
- Los comentarios vienen encabezados por la almohadilla (#) y continúan hasta el final de la línea.
- Es sensible a mayúsculas y minúsculas, todas las propiedades (palabras reservadas) de la especificación deben ir en minúsculas y terminar en dos puntos (:).
- Las propiedades requieren líneas independiente, su valor puede ir a continuación en la misma línea (precedido por un espacio) o en múltiples líneas (con sangrado)
- Las descripciones textuales pueden ser multilínea y admiten el dialecto CommonMark de Markdown para una representación de texto enriquecido. El HTML es compatible en la medida en que lo proporciona CommonMark (Bloques HTML en la Especificación 0.27 de CommonMark).
- $ref permite sustituir, reutilizar y enlazar una definición local con una externa.

CommonMark de Markdown

- Requiere doble y triple salto de línea para saltos de párrafos y cierre de bloques. Dos espacios al final de la línea lo convierte en salto de línea.
- Regla horizontal (separador): ---
- Énfasis: *cursiva* **negrita** ***cursiva y negrita***
- Enlaces: <http://www.example.com> [texto](http://www. example.com)
- Imágenes: ![Image](http://www.example.com/logo.png "icon")
- Citas: > Texto de la cita con sangría
- Bloques de códigos: `Encerrados entre tildes graves`
- Listas: Dos espacios en blanco por nivel de sangrado.
  + Listas desordenadas 1. Listas ordenadas
- Encabezado: dos líneas debajo del texto, añadir cualquier número de caracteres = para el nivel de título 1, <h1> … <h6> (el # es interpretado como comentario).

Estructura básica

- Un documento de OpenAPI puede estar compuesto por un solo documento o dividirse en múltiples partes conectadas a discreción del usuario. En el último caso, los campos $ref deben utilizarse en la especificación para hacer referencia a esas partes.
- Se recomienda que el documento raíz de OpenAPI se llame: openapi.json u openapi.yaml.
- La especificación de la API se puede dividir en 3 secciones principales:
  - Meta información
  - Elementos de ruta (puntos finales):
    - Parámetros de las solicitud
    - Cuerpo de las solicitud
    - Respuestas
  - Componentes reutilizables:
    - Esquemas (modelos de datos)
    - Parámetros
    - Respuestas
    - Otros componentes



Estructura básica

```
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in ...
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Estructura básica (cont)

```
components:
  schemas:
    User:
      properties:
        id:
          type: integer
        name:
          type: string
      # Both properties are required
      required:
        - id
        - name
  securitySchemes:
      BasicAuth:
        type: http
        scheme: basic
security:
  - BasicAuth: []
```


Prologo

- Cada definición de API debe incluir la versión de la especificación OpenAPI en la que se basa el documento en la propiedad openapi.
- La propiedad info contiene información de la API:
  - title es el nombre de API.
  - description es información extendida sobre la API.
  - version es una cadena arbitraria que especifica la versión de la API (no confundir con la revisión del archivo o la versión del openapi).
  - también admite otras palabras clave para información de contacto (nombre, url, email), licencia (nombre, url), términos de servicio (url) y otros detalles.
- La propiedad servers especifica el servidor API y la URL base. Se pueden definir uno o varios servidores (elementos precedidos por -).
- Con la propiedad externalDocs se puede referenciar la documentación externa adicional.

Rutas

- La sección paths define los puntos finales individuales (rutas) en la API y los métodos (operaciones) HTTP admitidos por estos puntos finales.
- Las ruta es relativa a la ruta del objeto Server.
- Los parámetros de la ruta se pueden usar para aislar un componente específico de los datos con los que el cliente está trabajando. Los parámetros de ruta son parte de la ruta y se expresan entre llaves (/users/{userId}), participan en la jerarquía de la URL y, por lo tanto, se apilan secuencialmente. Los parámetros de ruta deben describirse obligatoriamente en parameters (común para todas las operaciones) o a nivel de operación individual.
- No puede haber dos rutas iguales o ambiguas, que solo se diferencian por el parámetro de ruta.
- La definición de la ruta pude tener con un resumen (summary) y una descripción (description).
- Una ruta debe contar con un conjunto de operaciones, al menos una.
- Opcionalmente, servers permite dar una matriz alternativa de server que den servicio a todas las operaciones en esta ruta.

Rutas

```
'/users/{id}/roles':
  get:
    summary: Returns a list of users's roles.
    operationId: getDirecciones
    parameters:
      - in: path
        name: id
        description: User ID
        required: true
        schema:
          type: number
      - in: query
        name: size
        schema:
          type: string
          enum: [long, medium, short]
        required: true
      - in: query
        name: page
        schema:
          type: integer
          minimum: 0
          default: 0
```

```
responses:
  '200':
    description: List of roles
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Roles'
  '400':
    description: Bad request. User ID must be an
      integer and larger than 0.
  '401':
    description: Authorization information is missing
      or invalid.
  '404':
    description: A user with the specified ID was not
      found.
  '5XX':
    description: Unexpected error.
  default:
    description: Default error sample response
```

Operaciones

- Describe una única operación de API en una ruta y se identifica con el nombre del método HTTP: get, put, post, delete, options, head, patch, trace.
- Una definición de operación puede incluir un breve resumen de lo que hace (summary), una explicación detallada del comportamiento (description), una referencia a documentación externa adicional (externalDocs), un identificador único para su uso en herramientas y bibliotecas (operationId) y si está obsoleta y debería dejar de usarse (deprecated).
- Las operaciones pueden tener parámetros pasados a través de la ruta URL (/users/{userId}), cadena de consulta (/users?role=admin), encabezados (X-CustomHeader: Value) o cookies (Cookie: debug=0).
- Si la petición (POST, PUT, PATCH) envía un cuerpo en la solicitud (body), la propiedad requestBody permite describir el contenido del cuerpo y el tipo de medio.
- Para cada las respuestas de la operación, se pueden definir los posibles códigos de estado y el schema del cuerpo de respuesta. Los esquemas pueden definirse en línea o referenciarse mediante $ref. También se pueden proporcionar ejemplos para los diferentes tipos de respuestas.

Parámetros

- Un parámetro único se define mediante una combinación de nombre (name) y ubicación (in: "query", "header", "path" o "cookie") en la propiedad parameters.
- Opcionalmente puede ir acompañado por una breve descripción del parámetro (description), si es obligatorio (required), si permite valores vacíos (allowemptyvalue) y si está obsoleto y debería dejar de usarse (deprecated).
- Las reglas para la serialización del parámetro se especifican dos formas:
  - Para los escenarios más simples, con schema y style se puede describir la estructura y la sintaxis del parámetro.
  - Para escenarios más complejos, la propiedad content puede definir el tipo de medio y el esquema del parámetro.
- Un parámetro debe contener la propiedad schema o content, pero no ambas.
- Se puede proporcionar un example o examples pero debe seguir la estrategia de serialización prescrita para el parámetro.

Parámetros

```
paths:
  /users:
    get:
      description: Returns a list of users
      parameters:
        - name: rows
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: page
          in: query
          description: Specifies the page number of the users to be displayed
          schema:
            type: integer
```


Cuerpo de la solicitud

- En versiones anteriores, el cuerpo de la solicitud era un parámetro mas in: body.
- Actualmente se utiliza la propiedad requestBody con una breve descripción (description) y si es obligatorio para la solicitud (required), ambas opcionales.
- La descripción del contenido (content) es obligatoria y se estructura según los tipos de medios que actúan como identificadores. Para las solicitudes que coinciden con varias claves, solo se aplica la clave más específica (text/plain → text/* → */*).
- Por cada tipo de medio se puede definir el esquema del contenido de la solicitud (schema), uno (example) o varios (examples) ejemplos y la codificación (encoding).
- El requestBody sólo se admite en métodos HTTP donde la especificación HTTP 1.1 RFC7231 haya definido explícitamente semántica para cuerpos de solicitud.

Cuerpo de la solicitud

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - username
              properties:
                username:
                  type: string
                password:
                  type: string
                  format: password
                name:
                  type: string
```


Respuestas

- Es obligaría la propiedad responses con la lista de posibles respuestas que se devuelven al ejecutar esta operación.
- No se espera necesariamente que la documentación cubra todos los códigos de respuesta HTTP posibles porque es posible que ni se conozcan de antemano. Sin embargo, se espera que cubra la respuesta de la operación cuando tiene éxito y cualquier error previsto.
- La posibles respuestas se identifican con el código de respuesta HTTP. Con default se puede definir las respuesta por defecto para todos los códigos HTTP que no están cubiertos por la especificación individual.
- La respuesta cuenta con una breve descripción de la respuesta (description) y, opcionalmente, el contenido estructurado según los tipos de medios (content), los encabezados (headers) y los enlaces de operaciones que se pueden seguir desde la respuesta (links).

Respuestas

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody: # …
      responses:
        '201':
          description: Successfully created a new user
        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  code:
                    type: integer
                  message:
                    type: string
```


Etiquetas

- Las etiquetas son metadatos adicionales que permiten organizar la documentación de la especificación de la API y controlar su presentación. Las etiquetas se pueden utilizar para la agrupación lógica de operaciones por recursos o cualquier otro calificador. El orden de las etiquetas se puede utilizar para reflejar un orden en las herramientas de análisis.
- Cada nombre de etiqueta en la lista debe ser único (name) y puede ir acompañado por una explicación detallada (description) y una referencia a documentación externa adicional (externalDocs).
- Las etiquetas se pueden declarar en la propiedad tags del documento:

```
tags:
  - name: security-resource
    description: Gestión de la seguridad
```

Etiquetas

- Las etiquetas se aplican en la propiedad tags de las operaciones:

```
paths:
  /users:
    get:
      tags:
        - security-resource
  /roles:
    get:
      tags:
        - security-resource
        - read-only-resource
```

- No es necesario declarar todas las etiquetas, se pueden usar directamente pero no se podrá dar información adicional y se mostraran ordenadas al azar o según la lógica de las herramientas.

Componentes

- La propiedad global components permite definir las estructuras de datos comunes utilizadas en la especificación de la API: Contiene un conjunto de objetos reutilizables para diferentes aspectos de la especificación.
- Todos los objetos definidos dentro del objeto de componentes no tendrán ningún efecto en la API a menos que se haga referencia explícitamente a ellos desde propiedades fuera del objeto de componentes.
- La sección components dispones de propiedades para schemas, responses, parameters, examples, requestBodies, headers, securitySchemes, links y callbacks.
- Se puede hacer referencia a ellos con $ref cuando sea necesario. $ref acepta referencias internas con # o externas con el nombre de un fichero. La referencia debe incluir la trayectoria para encontrar el elemento referenciado:

  $ref: '#/components/schemas/Rol'

  $ref: responses.yaml#/404Error
- El uso de referencias permite la reutilización de elementos ya definidos, facilitando la mantenibilidad y disminuyendo sensiblemente la longitud de la especificación, por lo que se deben utilizar extensivamente. Las referencias no interfieren con la presentación en el UI.

Esquemas de datos

- Los schemas definen los modelos de datos consumidos y devueltos por la API.
- Los tipos de datos OpenAPI se basan en un subconjunto extendido del JSON Schema Specification Wright Draft 00 (también conocido como Draft 5).
- Los tipos base son string, number, integer, boolean, array y object.
- Con la propiedad format se pueden especificar otros tipos especiales partiendo de los tipos base: long, float, double, byte, binary, date, dateTime, password.
- Los tipos array se definen como una colección de ítems y en dicha propiedad se define el tipo y la estructura de los elementos que lo componen. Los objetos son un conjunto de propiedades, cada una definida dentro de properties.
- Cada tipo y propiedad se identifica por un nombre que no debe estar repetido en su ámbito.
- Cada propiedad puede definir description, default, minimum, maximum, maxLength, minLength, pattern, required, readOnly, …
- Para una propiedad se pueden definir varios tipos (tipos mixtos o unión).
- Los tipos pueden hacer referencia a otros tipos.

Tipos de datos

type	format	Comentarios
boolean		Booleanos: true y false
integer	int32	Enteros con signo de 32 bits
integer	int64	Enteros con signo de 64 bits (también conocidos como largos)
number	float	Reales cortos
number	double	Reales largos
string		Cadenas de caracteres
string	password	Una pista a las IU para ocultar la entrada.
string	date	Según lo definido por full-date RFC3339 (2018-11-13)
string	date-time	Según lo definido por date-time- RFC3339 (2018-11-13T20:20:39+00:00)
string	byte	Binario codificados en base64
string	binary	Binario en cualquier secuencia de octetos
array		Colección de items
object		Colección de properties

Propiedades de los objetos de esquema

- type: integer, number, boolean, string, array, object
- format: long, float, double, byte, binary, date, dateTime, password
- title: Nombre a mostrar en el UI
- description: Descripción de su uso
- maximum: Valor máximo
- exclusiveMaximum: Valor menor que
- minimum: Valor mínimo
- exclusiveMinimum: Valor mayor que
- multipleOf: Valor múltiplo de
- maxLength: Longitud máxima
- minLength: Longitud mínima
- pattern: Expresión regular del patrón
- deprecated: Si está obsoleto y debería dejar de usarse
- nullable: Si acepta nulos
- default: Valor por defecto
- enum: Lista de valores con nombre
- example: Ejemplo de uso
- externalDocs: referencia a documentación externa adicional
- items: Definición de los elementos del array
- maxItems: Número máximo de elementos
- minItems: Número mínimo de elementos
- uniqueItems: Elementos únicos
- properties: Definición de las propiedades del objeto,
- maxProperties: Número máximo de propiedades
- minProperties: Número mínimo de propiedades
- readOnly: propiedad de solo lectura
- writeOnly: propiedad de solo escritura
- additionalProperties: permite referenciar propiedades adicionales
- required: Lista de propiedades obligatorias

Modelos de entrada y salida

```
components:
  schemas:
    Roles:
      type: array
      items:
        $ref: '#/components/schemas/Rol'
    Rol:
      type: object
      description: Roles de usuario
      properties:
        rolId:
          type: integer
          format: int32
          minimum: 0
          maximum: 255
        name:
          type: string
          maxLength: 20
        description:
          type: string
        last_updated:
          type: string
          format: dateTime
          readOnly: true
        level:
            type: string
            description: Nivel de permisos
            enum:
            - high
            - normal
            - low
            default: normal
      required:
      - rolId
      - name
```

Autenticación

- La propiedad securitySchemes de components y la propiedad security del documento se utilizan para describir y establecer los métodos de autenticación utilizados en la API.
- securitySchemes define los esquemas de seguridad que pueden utilizar las operaciones. Los esquemas admitidos son la autenticación HTTP, una clave API (ya sea como encabezado, parámetro de cookie o parámetro de consulta), los flujos comunes de OAuth2 (implícito, contraseña, credenciales de cliente y código de autorización) tal y como se define en RFC6749 y OpenID Connect Discovery. Cada esquema cuenta con un identificador, un tipo (type: "apiKey", "http", "oauth2", "openIdConnect") y opcionalmente puede ir acompañado por una breve descripción (description).
- Según el tipo seleccionado será obligatorio:
  - apiKey: ubicación (in: "query", "header" o "cookie") y su nombre (name) de parámetro, encabezado o cookie.
  - http: esquema de autorización HTTP que se utilizará en el encabezado Authorization (scheme): Basic, Bearer, Digest, OAuth, … y, si es Bearer, prefijo del token de portador (bearerFormat).
  - openIdConnect: URL de OpenId Connect para descubrir los valores de configuración de OAuth2 (openIdConnectUrl).
  - oauth2: objeto que contiene información de configuración para los tipos de flujo admitidos (flows).
- La propiedad security enumera los esquemas de seguridad que se pueden utilizar en la API.

Autenticación

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    JWTAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
    ApiKeyAuth:
      type: apiKey
      name: x-api-key
      in: header
    ApiKeyQuery:
      type: apiKey
      name: api-key
      in: query
security:
  - ApiKeyAuth: []
  - ApiKeyQuery: []
```

Ejemplos

- Los ejemplos son fundamentales para la correcta comprensión de la documentación. La especificación permite proporcionar uno (example) o varios (examples) ejemplos asociados a las estructuras de datos.
- Por cada uno se puede dar un resumen del ejemplo (summary), una descripción larga (description), el juego de valores de las propiedades de la estructura (value) o una URL que apunta al ejemplo literal para ejemplos que no se pueden incluir fácilmente en documentos JSON o YAML (externalValue). value y externalValue son mutuamente excluyentes. Cuando son varios ejemplos deber estar identificados por un nombre único.

```
examples:
  first-page:
    summary: Primera página
    value: 0
  second-page:
    summary: Segunda página
    value: 1
```

- Los ejemplos pueden ser utilizados automáticamente por las herramientas de UI y de generación de pruebas.

Ecosistema Swagger

- Swagger Open Source Tools (https://swagger.io/)
  - Swagger UI: Generar automáticamente la documentación desde la definición de OpenAPI para la interacción visual y un consumo más fácil.
  - Swagger Editor: Diseñar APIs en un potente editor de OpenAPI que visualiza la definición y proporciona comentarios de errores en tiempo real.
  - Swagger Codegen: Crear y habilitar el consumo de su API generando la fontanería del servidor y el cliente.
- Swagger Pro Tools
  - SwaggerHub: La plataforma de diseño y documentación para equipos e individuos que trabajan con la especificación OpenAPI.
  - Swagger Inspector: La plataforma de pruebas y generación de documentación de las APIs
- https://openapi.tools/

OpenApi

- Dependencias:
  - swagger-jsdoc: módulo que lee las anotaciones JSDoc marcadas con @swagger de el código fuente y genera una especificación OpenAPI (Swagger).
  - swagger-ui-express: módulo permite servir documentos OpenAPI con swagger-ui desde Express.
  - express-openapi-validator: módulo middleware para Express que valida automáticamente las solicitudes y respuestas de la API mediante una especificación de OpenAPI 3.
- Instalación:
  - npm i swagger-ui-express swagger-jsdoc express-openapi-validator

swagger-jsdoc

```
/**
 * @swagger
 * components:
 *  schemas:
 *    Login:
 *      description: Credenciales de autenticación
 *      type: object
 *      required:
 *        - username
 *        - password
 *      properties:
 *        name:
 *          type: string
 *        password:
 *          type: string
 *          format: password
 *    RespuestaLogin:
 *      type: object
 *      title: Respuesta Login
 *      properties:
 *        success:
 *          type: boolean
 *        token:
 *          type: string
 *        name:
 *          type: string
 *        roles:
 *          type: array
 *          items:
 *            type: string
 */
/**
 * @swagger
 *
 * /login:
 *   post:
 *     tags: [ autenticación ]
 *     summary: Iniciar sesión
 *     requestBody:
 *       content:
 *         application/json:
 *           schema:
 *             $ref: "#/components/schemas/Login"
 *       required: true
 *     responses:
 *       "200":
 *         headers:
 *           Set-Cookie:
 *             schema:
 *               type: string
 *         description: "Resultado de la autenticación"
 *         content:
 *           application/json:
 *             schema:
 *               $ref:
"#/components/schemas/RespuestaLogin"
 *       "400":
 *         $ref: "#/components/responses/BadRequest"
 */
router.post('/login', async function (req, res, next) {
```

swagger-jsdoc

```
const swaggerJsdoc = require('swagger-jsdoc');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Microservicios',
      version: '1.0.0',
    },
  },
  apis: ['./src/routes*.js'],
};
const openapiSpecification = swaggerJsdoc(options);

app.all('/api-docs/v1/openapi.json', (req, res) => res.json(openapiSpecification));
```


swagger-ui-express

- Documento estático:

```
const swaggerUi = require('swagger-ui-express');
const swaggerDocument = require('./swagger.json');
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```

- Documento dinámico (swagger-jsdoc)

```
const openapiSpecification = swaggerJsdoc(options);
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(openapiSpecification));
```

- Enlace al documento

```
var options = {
   swaggerOptions: {
      url: "/api-docs/v1/openapi.json",
   },
}
app.all('/api-docs/v1/openapi.json', (req, res) => res.json(openapiSpecification));
app.use('/api-docs', swaggerUi.serveFiles(null, options), swaggerUi.setup(null, options));
```

express-openapi-validator

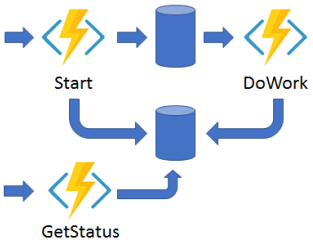
```
const OpenApiValidator = require('express-openapi-validator');

const openapiSpecification = swaggerJsdoc(options);

app.use(
  OpenApiValidator.middleware({
    apiSpec: openapiSpecification,
    validateRequests: true, // (default)
    validateResponses: true, // false by default
    ignoreUndocumented: true,
    formats: [
      { name: 'nif', type: 'string', validate: (v) => validator.isIdentityCard(v, 'ES') },
    ]
  })
)
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ message: err.message, errors: err.errors,});
});
```

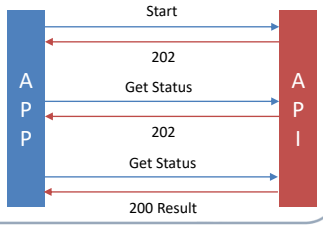

WebHooks

- Los webhooks son eventos HTTP que desencadenan acciones. Su nombre se debe a que funcionan como «enganches» de los programas en Internet y casi siempre se utilizan para la comunicación entre sistemas. Son la manera más sencilla de obtener un aviso cuando algo ocurre en otro sistema y para el intercambio de datos entre aplicaciones web, permitiendo las llamada asíncronas en HTTP.
- Un webhook es una retro llamada HTTP, una solicitud HTTP GET/POST insertada en una página web, que interviene cuando ocurre algo (una notificación de evento a través de HTTP GET/POST).
- Los webhooks se utilizan para las notificaciones en tiempo real (con los datos del evento como parámetros o en cuerpo en JSON o XML) a una determinada dirección http:// o https://, que puede:
  - almacenar los datos del evento en JSON o XML
  - generar una respuesta que permita actualizarse al sistema donde se produce el evento
  - ejecutar un proceso en el sistema receptor del evento (Ej: enviar un correo electrónico)
- Los webhook están pensados para su utilización desde páginas web y sus diferentes consumidores: navegadores, correo electrónico, webapps, …
- Un ejemplo típico es su utilización en correos electrónico de marketing para notificar al servidor que debe enviar un nuevo correo electrónico porque el usuario a abierto el mensaje.
- Pueden considerarse una versión especializada y simplificada de los servicios REST (solo GET/POST).

WebHooks

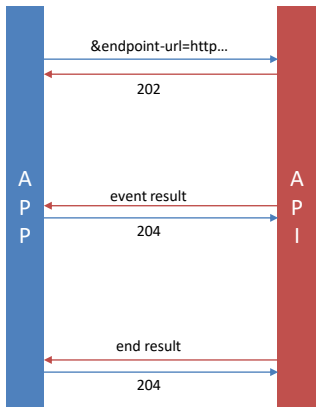
- El patrón Async HTTP APIs soluciona el problema de coordinar el estado de las operaciones de larga duración con los clientes externos. Una forma habitual de implementar este patrón es que un punto de conexión HTTP desencadene la acción de larga duración. A continuación, el cliente se redirige a un punto de conexión de estado al que sondea para saber el estado actual del proceso y cuando finaliza la operación.
- Un endpoint HTTP desencadena el proceso con la acción de larga duración y devuelve inmediatamente un 202 si la petición es correcta y, opcionalmente, como carga útil las URLs de los endpoints de estado y control. Las peticiones incorrectas recibirán el 4XX apropiado.
- El cliente sondea periódicamente el endpoint de estado y obtiene:
  - 202 con el estado actual del proceso como carga útil mientras el proceso este en marcha.
  - 200 con el estado final (correcto o fallido) del proceso como carga útil cuando haya finalizado la operación.
- El endpoint HTTP desencadenador puede suministrar endpoints adicionales para pausar, reanudar, cancelar/terminar, reiniciar o enviar eventos al proceso en marcha.





WebHooks

- El patrón Reverse API soluciona el problema del sondeo periódico, las API inversas invierten esta situación, para que sea la API invocada la que notifique automáticamente cuando se ha producido un determinado evento. El cliente debe crear su propia API para recibir las notificaciones.
- Al realizar la petición al endpoint HTTP desencadenador, se suministra un endpoint propio para que el proceso desencadenado pueda notificar cuándo ocurre algo de interés. Le da la vuelta a la comunicación, el cliente pasa a ser servidor y el servidor a cliente. El desencadenador devuelve inmediatamente un 202 si la petición es correcta o el 4XX apropiado se es incorrecta.
- Este esquema de funcionamiento tiene muchas ventajas en ambos extremos:
  - **Ahorro de recursos y tiempo**: con el sondeo se harán muchas llamadas "para nada", que no devolverán información relevante. Los dos extremos gastan recursos para hacer y responder a muchas llamadas que no tienen utilidad alguna (no nos llame, ya les llamaremos).
  - **Eliminación de los retrasos**: la aplicación usuaria recibirá una llamada en el momento exacto en el que se produce y no tendrá retrasos al próximo sondeo.
  - **Velocidad de las llamadas**: generalmente la llamada que se hace a un webhook es muy rápida porque solo se envía una pequeña información sobre el evento y se suele procesar asíncronamente. Muchas veces ni siquiera se espera por el resultado: se hace una llamada del tipo "fire and forget" (o sea, dispara y olvídate), pues se trata de notificar el evento y listo.



SEGURIDAD

Conceptos

- La autenticación es un proceso en el que un usuario o una aplicación se identifica proporcionando credenciales que después se comparan con las almacenadas en un sistema operativo, base de datos, aplicación o recurso para validar que es realmente quién asegura ser. La autenticación puede crear una o varias identidades para el usuario o aplicación autenticado.
- La autorización se refiere al proceso que determina lo que una identidad puede hacer en función a los permisos otorgados a una identidad concreta sobre un recurso concreto. La autorización es ortogonal e independiente de la autenticación. Sin embargo, la autorización requiere un mecanismo de autenticación. La autorización puede utilizar un modelo basado en roles, sencillo y declarativo, o un modelo avanzado basado en directivas y evidencias.

Autenticación por formularios

- Ventajas:
  - Usa su propia infraestructura que se puede personalizar
  - Admite todos los tipos de clientes (cualquier S.O. y dominio, incluyendo la autenticación delegada)
- Desventajas:
  - Menor nivel de seguridad
  - Se basa en cookies
  - Requiere comunicaciones seguras dado que el usuario y la contraseña se transmiten en abierto
- Recomendable para su uso en Internet.

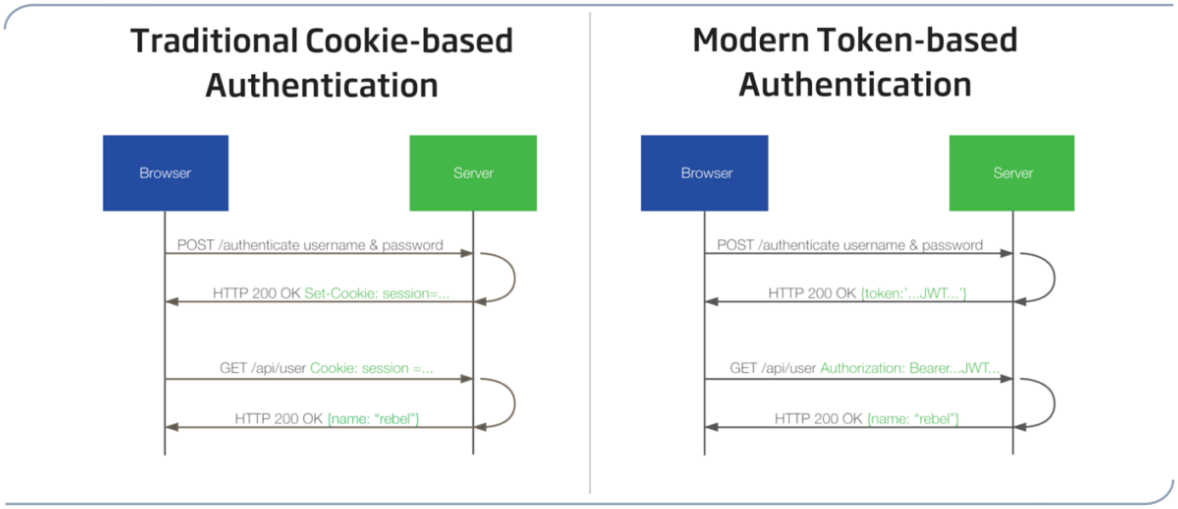
Multi-factor Authentication (MFA)

- Multi-factor Authentication (MFA) es un proceso en el que se solicita un usuario durante un evento de inicio de sesión para otras formas de identificación. Este mensaje puede indicar un código de un teléfono móvil, usar una clave FIDO2 o proporcionar un análisis de huellas digitales. Cuando se requiere una segunda forma de autenticación, se mejora la seguridad. Un atacante no obtiene ni duplica fácilmente el factor adicional.
- MFA requiere al menos dos o más tipos de pruebas para una identidad como algo que conoce, algo que posee o validación biométrica para que el usuario se autentique. La autenticación en dos fases (2FA) es como un subconjunto de MFA, pero la diferencia es que MFA puede requerir dos o más factores para demostrar la identidad. MFA con SMS aumenta la seguridad de forma masiva en comparación con la autenticación de contraseña (factor único). Sin embargo, ya no se recomienda usar SMS como segundo factor. Existen demasiados vectores de ataque conocidos para este tipo de implementación.

Autenticación en dos fases

- MFA con TOTP, un algoritmo de contraseña de un solo uso, es una implementación. Se puede usar junto con cualquier aplicación de autenticador compatible, lo que incluye:
  - Aplicación Microsoft Authenticator
  - Aplicación de Google Authenticator
- Una aplicación autenticadora proporciona un código de 6 a 8 dígitos que los usuarios deben escribir después de confirmar su nombre de usuario y contraseña.
- El enfoque recomendado del sector para 2FA es TOTP frente SMS.
- Normalmente, una aplicación autenticadora se instala en un smartphone.
- Se puede habilitar la generación de código QR para las aplicaciones de TOTP Authenticator.

Autenticación



Single Sign On (SSO)

- El Single SingOn (SSO o inicio de sesión única) es un patrón arquitectónico para permitir a los usuarios el acceso a varias aplicaciones mediante una solo autenticación, utilizando para ello, un proveedor de autenticación en común. SSO permite delegar el proceso de autenticación a una entidad externa, la cual tiene como única responsabilidad autenticar que el usuario es quien dice ser, una vez autenticado recibe un token de identidad.
- Esto quieres decir que un usuario podría entrar a varias aplicaciones sin necesidad de tener que autenticarse en cada una, en su lugar, solo requerirá autenticarse la primera vez en cualquiera de las aplicaciones y posteriormente podrá acceder al resto si pasar por el proceso de autenticación.
- Hay varios tipos principales de SSO, también llamados reduced sign-on systems ("sistemas de autenticación reducida").
  - Enterprise SSO (E-SSO), también llamado legacy sso, funciona para una autenticación primaria, interceptando los requisitos de login presentados por las aplicaciones secundarias para completar los mismos con el usuario y contraseña. Los sistemas E-SSO permiten interactuar con sistemas que pueden deshabilitar la presentación de la pantalla de login.
  - Web SSO (Web-SSO), también llamado gestión de acceso web (web access management, Web-AM o WAM) trabaja solamente con aplicaciones y recursos accedidos vía web.
  - Kerberos es un método popular de externalizar la autenticación de los usuarios. Los usuarios se registran en el servidor Kerberos y reciben un tique, luego las aplicaciones cliente lo presentan para obtener acceso.
  - Identidad federada es una nueva manera de enfrentar el problema de la autenticación, también para aplicaciones Web. Utiliza protocolos basados en estándares para habilitar que las aplicaciones puedan identificar los clientes sin necesidad de autenticación redundante.
  - OpenID es un proceso de SSO distribuido y descentralizado donde la identidad se compila en un Localizador Uniforme de Recursos (URL) que cualquier aplicación o servidor puede verificar.

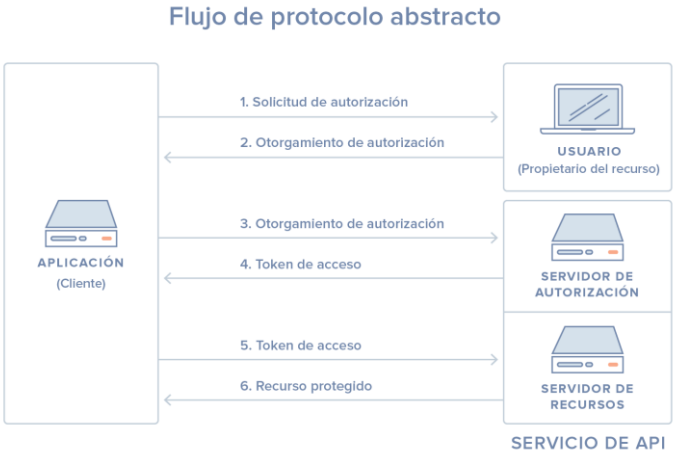
Ventajas y Desventajas del SSO

- Ventajas
  - Acceso rápido a las aplicaciones: Con el SSO, los usuarios pueden acceder rápidamente a múltiples aplicaciones y servicios con una sola autenticación.
  - Simplificación de la experiencia del usuario.
  - Mayor seguridad: Al utilizar el SSO, se puede implementar una autenticación más robusta y segura.
  - Administración simplificada: El SSO simplifica la administración de usuarios y contraseñas en un entorno empresarial. Los administradores pueden gestionar de manera centralizada las cuentas de usuario y los permisos de acceso, lo que facilita la incorporación y desactivación de usuarios en los
- Desventajas
  - Vulnerabilidad única: Si el SSO se ve comprometido, todas las aplicaciones y servicios vinculados a él también pueden estar en riesgo.
  - Dependencia de la disponibilidad del sistema SSO: Si el sistema SSO experimenta una interrupción o se vuelve inaccesible, los usuarios podrían perder el acceso a todas las aplicaciones y servicios vinculados. Es un potencial cuello de botellas.
  - Complejidad de implementación: La implementación de un sistema SSO puede ser compleja, especialmente en entornos empresariales con múltiples aplicaciones y sistemas.
  - Privacidad y confianza: Al utilizar el SSO, los usuarios deben confiar en que su proveedor de SSO protegerá adecuadamente sus datos personales y de inicio de sesión.

OAuth 2

- OAuth 2 es un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a los recursos de usuario en un servicio HTTP, como Facebook, GitHub y Google. Delega la autenticación del usuario al servicio que aloja la cuenta del mismo y autoriza a las aplicaciones de terceros el acceso a dicha cuenta de usuario. OAuth define cuatro roles:
  - Propietario del recurso: Una entidad capaz de otorgar acceso a un recurso protegido. Cuando el propietario del recurso es una persona, se le conoce como usuario final.
  - Servidor de recursos: El servidor que aloja los recursos protegidos, capaz de aceptar y responder a solicitudes de recursos protegidos utilizando tokens de acceso.
  - Cliente: Una aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización. El término "cliente" no implica ninguna característica de implementación particular.
  - Servidor de autorizaciones: El servidor que emite tokens de acceso al cliente después de haber realizado correctamente la autenticación y validar la concesión del propietario del recurso.

OAuth 2



Patrones de diseño

- Patrón: Token de acceso
  - Ha aplicado la arquitectura de microservicios y los patrones de API Gateway. La aplicación consta de numerosos servicios. La puerta de enlace API es el único punto de entrada para las solicitudes de los clientes. Autentica las solicitudes y las reenvía a otros servicios, que a su vez podrían invocar otros servicios. ¿Cómo comunicar la identidad del solicitante a los servicios que tramitan la solicitud? El API Gateway autentica la solicitud y pasa un token de acceso (por ejemplo, JSON Web Token) que identifica de forma segura al solicitante en cada solicitud a los servicios. Un servicio puede incluir el token de acceso en las solicitudes que realiza a otros servicios.
- Patrón: Valet Key
  - Usa un token que proporciona a los clientes acceso directo restringido a un recurso específico, con el fin de descargar la transferencia de datos desde la aplicación. Esto es especialmente útil en aplicaciones que usan sistemas o colas de almacenamiento hospedado en la nube, ya que puede minimizar los costes y maximizar la escalabilidad y el rendimiento.
- Patrón: Federated Identity
  - La autenticación se delega a un proveedor de identidad externo. Esto puede simplificar el desarrollo, minimizar los requisitos de administración de usuarios y mejorar la experiencia del usuario de la aplicación.

Autenticación sin estado basada en tokens

- Para solucionar los problemas de sobrecarga y escalabilidad provocados la autenticación basada en sesiones y cookies, surge la autenticación sin estado (stateless). Esto significa que el servidor no va a almacenar ninguna información, ni tampoco la sesión.
- Cuando el usuario se autentica con sus credenciales o cualquier otro método, en la respuesta recibe un token (access token) y, opcionalmente, un refresh token. El token es una cadena encriptada firmada, para evitar alteraciones y ser confiable, con una fecha de expiración corta para evitar vulnerabilidades de seguridad.
- A partir de ese momento, todas las peticiones que se hagan al API llevarán este token en una cabecera HTTP de modo que el servidor pueda identificar qué usuario hace la petición y, una vez verificado el token, confiar en las credenciales suministradas sin necesidad de buscar en base de datos ni en ningún otro sistema de almacenamiento o agente externo.
- Con este enfoque, la aplicación pasa a ser escalable, ya que es el propio cliente el que almacena su información de autenticación, y no el servidor. Así las peticiones pueden llegar a cualquier instancia del servidor y podrá ser atendida sin necesidad de sincronizaciones. Así mismo, diferentes plataformas podrán usar el mismo API. Además se incrementa la seguridad, evitando vulnerabilidades CSRF, al no existir sesiones.
- El refresh token es una identidad verificada y se usa para generar un nuevo access token cuando este expira. Típicamente, si el access token tiene fecha de expiración corta, una vez que caduca, el usuario tendría que autenticarse de nuevo para obtener un nuevo access token. Con el refresh token, que identifica al usuario y tiene una expiración mas generosa, este paso se puede saltar y con una petición al API obtener un nuevo access token que permita al usuario seguir accediendo de forma transparente a los recursos de la aplicación.

Bearer Authentication

- La autenticación de portador (también llamada token de autenticación) es un esquema de autenticación HTTP que involucra tokens de seguridad llamados tokens de portador (Bearer). El nombre "Autenticación de portador" puede entenderse como "dar acceso al portador de este token". El token portador es una cadena encriptada, generalmente generada por el servidor en respuesta a una solicitud de inicio de sesión (Access token). El cliente debe enviar este token en el encabezado Authorization al realizar solicitudes a recursos protegidos:

  Authorization: Bearer <token>
- El esquema de autenticación Bearer se creó originalmente como parte de OAuth 2.0 en RFC 6750, pero a veces también se usa solo. De manera similar a la autenticación básica, la autenticación de portador solo debe usarse a través de HTTPS (SSL).

JWT: JSON Web Tokens

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier dato entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo, si tenemos un servicio de envío de email, otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

https://jwt.io

Tokens

- Los tokens son una serie de caracteres cifrados y firmados con una clave compartida entre servidor OAuth y el servidor de recurso o para mayor seguridad mediante clave privada en el servidor OAuth y su clave pública asociada en el servidor de recursos, con la firma el servidor de recursos el capaz de comprobar la autenticidad del token sin necesidad de comunicarse con él.
- Se componen de tres partes separadas por un punto: una cabecera con el algoritmo hash utilizado y tipo de token, un documento JSON con datos y una firma de verificación.
- El hecho de que los tokens JWT no sea necesario persistirlos en base de datos elimina la necesidad de tener su infraestructura, como desventaja es que no es tan fácil de revocar el acceso a un token JWT y por ello se les concede un tiempo de expiración corto.

- La infraestructura requiere varios elementos configurables de diferentes formas:
  - El servidor OAuth que realiza la autenticación y proporciona los tokens.
  - El servicio al que se le envía el token, es el que decodifica el token y decide conceder o no acceso al recurso.
  - En el caso de múltiples servicios con múltiples recursos es conveniente un gateway para que sea el punto de entrada de todos los servicios, de esta forma se puede centralizar las autorizaciones liberando a los servicios individuales.

Generar token

```
$ npm install jsonwebtoken --save

const jwt = require('jsonwebtoken')
const APP_SECRET = 'Es segura al 99%'
const AUTHENTICATION_SCHEME = 'Bearer ‘

app.post(DIR_API_AUTH + 'login', function (req, res) {
      // …
      let token = AUTHENTICATION_SCHEME + jwt.sign({
        usr: ele[PROP_USERNAME],
        name: ele.nombre,
        roles: ele.roles
      }, APP_SECRET, { expiresIn: '1h’ })
      res.status(200).end(token)
}
```

Middleware de autenticación

```
app.use(function (req, res, next) {
  res.locals.isAutenticated = false;
  if (!req.headers['authorization']) {
    next();
    return;
  }
  let token = req.headers['authorization'].substr(AUTHENTICATION_SCHEME.length)
  try {
    var decoded = jwt.verify(token, APP_SECRET);
    res.locals.isAutenticated = true;
    res.locals.usr = decoded.usr;
    res.locals.name = decoded.name;
    res.locals.roles = decoded.roles;
    next();
  } catch (err) {
    res.status(401).end();
  }
})
```


Cifrado asimétrico

```
const TokenRS256 = {
    generar: (usuario) => {
        let buff = Buffer.from(config.security.PRIVATE_KEY, 'base64');

        return jwt.sign({
            usr: usuario[config.security.PROP_USERNAME],
            name: usuario.nombre,
            roles: usuario.roles
        }, createPrivateKey({ key: buff, format: 'der', type: 'pkcs8' }), { issuer: 'MicroserviciosJWT', audience:
                    'authorization', algorithm: 'RS256', expiresIn: config.security.EXPIRACION_MIN + 'm' })
    },
    decode: (token) => {
        let buff = Buffer.from(config.security.PUBLIC_KEY, 'base64');
        return jwt.verify(token, createPublicKey({ key: buff, format: 'der', type: 'spki' }), { algorithms: ['RS256'] });
    }
}
```

Middleware: Autorización

```
module.exports.onlyAuthenticated = (req, res, next) => {
  if (req.method === 'OPTIONS') return next()
  if (!res.locals.isAuthenticated) return next(generateErrorByStatus(401))
  next()
}
module.exports.onlyInRole = (roles) => (req, res, next) => {
  if (req.method === 'OPTIONS') return next()
  if (!res.locals.isAuthenticated) return next(generateErrorByStatus(401))
  if (roles.split(',').some(role => res.locals.isInRole(role))) {
    next()
  } else {
    return next(generateErrorByStatus(403))
  }
}
module.exports.onlySelf = (_req, res, next) => {
  res.locals.onlySelf = true;
  next()
}
module.exports.readOnly = (req, res, next) => (!['GET', 'OPTIONS'].includes(req.method) && !res.locals.isAuthenticated) ?
          next(generateErrorByStatus(401)) : next()
```


Cifrado de claves

- Nunca se debe almacenar las contraseñas en texto plano, uno de los procesos básicos de seguridad contra robo de identidad es el cifrado de las claves de usuario.
- MD5, SHA1, SHA2, SHA3 y otras funciones hash de propósito general, están diseñadas para calcular un resumen de grandes cantidades de datos en el menor tiempo posible. Esto significa que son fantásticas para garantizar la integridad de los datos y absolutamente inútiles para almacenar contraseñas. Existen algoritmos hash deliberadamente diseñados para ser lentos y consumir un máximo de recursos para el cifrado de las claves.
- Bcrypt es una función de hashing de contraseñas diseñado por Niels Provos y David Maxieres, basado en el cifrado de Blowfish. Se usa por defecto en sistemas OpenBSD y algunas distribuciones Linux y SUSE. Lleva incorporado un valor llamado salt, que es un fragmento aleatorio que se usará para generar el hash y se guardará junto con ella. Así se evita que dos contraseñas iguales generen el mismo hash y los problemas que ello conlleva, por ejemplo, el ataque por fuerza bruta generando todas las posibles contraseñas y sus hash.
- Otro ataque relacionado es el de Rainbow table (tabla arcoíris, diccionario pre calculado de textos con sus hash asociadas), para buscar por hash la contraseña asociada. El salt cambia el hash asociado a una contraseña sea único. Aplicar sucesivamente el algoritmo varias iteraciones introduce una complejidad adicional.

Cifrado de claves

- Instalación:
  - npm install bcrypt
- Para auto generar el satl, con 10 iteraciones, y la hash asociada:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
async function encriptaPassword(password) {
   const salt = await bcrypt.genSalt(saltRounds)
   const hash = await bcrypt.hash(password, salt)
   console.log(hash)
   return hash
}
```

- Para comprobar que la contraseña recibida coincide con la hash almacenada:

```
async function verifyPassword(password, hash) {
   return await bcrypt.compare(password, hash)
}
```


Passport JS

https://www.passportjs.org/

- Passport es un middleware de autenticación compatible con Express para Node.js. El único propósito de Passport es autenticar las solicitudes, lo que hace a través de un conjunto extensible de complementos conocidos como estrategias. Las estrategias pueden ir desde la verificación de las credenciales del nombre de usuario y la contraseña, la autenticación delegada mediante OAuth (por ejemplo, a través de Facebook o Twitter) o la autenticación federada mediante OpenID.
- Passport no monta rutas ni asume ningún esquema de base de datos en particular, lo que maximiza la flexibilidad y permite que el desarrollador tome decisiones a nivel de aplicación.
- La API es simple: se proporciona a Passport una solicitud de autenticación y Passport proporciona ganchos para controlar lo que ocurre cuando la autenticación tiene éxito o falla.

Prevenir ataques de fuerza bruta contra la autorización

- Es esencial asegurarse de que los puntos finales de inicio de sesión estén protegidos para que los datos privados sean más seguros.
- Una técnica simple y poderosa es bloquear los intentos de autorización usando dos métricas:
  - El primero es el número de intentos fallidos consecutivos del mismo nombre de usuario y dirección IP.
  - El segundo es el número de intentos fallidos desde una dirección IP durante un largo período de tiempo. Por ejemplo, bloquear una dirección IP si hace 100 intentos fallidos en un día.
- El paquete rate-limiter-flexible proporciona herramientas para hacer que esta técnica sea fácil y rápida.

Protección ante XSRF

- Cross-Site Request Forgery (XSRF) explota la confianza del servidor en la cookie de un usuario.
- La protección puede establecerse a nivel de formulario y requerir solo intervención del servidor, envía un token al cliente en el formulario que verifica al recibir la devolución del formulario:

  ```
  <input type="hidden" name="XSRF-TOKEN" value="123456790ABCDEF">
  ```
- El mecanismo “Cookie-to-Header Token” para prevenir ataques XSRF.
  - El servidor debe establecer un token en una cookie de sesión legible en JavaScript, llamada XSRF-TOKEN, en la carga de la página o en la primera solicitud GET. En las solicitudes posteriores, el cliente debe incluir el encabezado HTTP X-XSRF-TOKEN con el valor recibido en la cookie.
  - El servidor puede verificar que el valor en la cookie coincida con el del encabezado HTTP y, por lo tanto, asegúrese de que sólo el código que se ejecutó en su dominio pudo haber enviado la solicitud.
  - El token debe ser único para cada usuario y debe ser verificable por el servidor. Para mayor seguridad se puede incluir el token en un resumen de la cookie de autenticación de su sitio.

XSRF middleware

- Implementación manual:

```
const xsrfToken = '123456790ABCDEF'
app.use(cookieParser())
function generateXsrfTokenCookie(res) {
   res.cookie('XSRF-TOKEN', xsrfToken, { httpOnly: false })
}
app.use(function (req, res, next) {
  if ('POST|PUT|DELETE|PATCH'.includes(req.method.toUpperCase() && req.cookies['XSRF-TOKEN'] !== req.headers['x-xsrf-token'] && req.cookies['XSRF-
      TOKEN'] !== xsrfToken ) {
    res.status(401).end('No autorizado.')
    return
  }
  next()
})
```

- Paquete csurf de Connect/Express

```
$ npm install csurf --save
var cors = require('csurf')
var csrfProtection = csrf({ cookie: true })
app.get('/form', csrfProtection, function (req, res) {
  res.render('send', { csrfToken: req.csrfToken() })
})
```


Prácticas recomendadas en producción

- https://owasp.org/www-project-top-ten/
- No usar versiones obsoletas o vulnerables de Express y asegurarse de que las dependencias sean seguras
  - Usar npm para administrar las dependencias de la aplicación es poderoso y conveniente. Pero los paquetes que se usan pueden contener vulnerabilidades de seguridad críticas que también podrían afectar la aplicación. La seguridad de la aplicación es tan fuerte como el "eslabón más débil" en sus dependencias.

    npm audit
- Usar TLS
  - Si la aplicación trata o transmite datos confidenciales, usar Transport Layer Security (TLS) para proteger la conexión y los datos. Esta tecnología encripta los datos antes de enviarlos del cliente al servidor, evitando así algunos ataques comunes (y fáciles). Aunque las solicitudes de Ajax y POST pueden no ser visiblemente obvias y parecer "ocultas" en los navegadores, su tráfico de red es vulnerable a la detección de paquetes y a los ataques de intermediarios.

Prácticas recomendadas en producción

- Usar casco (Helmet)
  - Helmet puede ayudar a proteger la aplicación de algunas vulnerabilidades web conocidas al configurar los encabezados HTTP de manera adecuada.
  - Helmet es una colección de varias funciones de middleware más pequeñas que establecen encabezados de respuesta HTTP relacionados con la seguridad.

```
npm install --save helmet
const helmet = require('helmet')
app.use(helmet())
```


Prácticas recomendadas en producción

- Siempre se debe filtrar, validar y desinfectar las entradas de usuario para protegerse contra los ataques de inyección de comandos y secuencias de comandos en sitios cruzados (XSS): express-validator, validator.js o express-sanitize-input.
- Defenderse contra los ataques de inyección de código SQL mediante el uso de consultas parametrizadas o declaraciones preparadas.
- Utilizar la herramienta de código abierto sqlmap para detectar vulnerabilidades de inyección SQL en la aplicación.
- Utilizar las herramientas nmap y sslyze para probar la configuración de los cifrados, claves y renegociación SSL, así como la validez del certificado.
- Usar safe-regex para asegurarse de que las expresiones regulares no sean susceptibles a ataques de denegación de servicio de expresiones regulares.

DESPLIEGUE

Despliegue

- El despliegue más simple posible
  1. Copiar (xcopy) todos los directorios y archivos de producción a una carpeta en el servidor, excluyendo ficheros de configuración, pruebas, node_modules, …
  2. Instalar todas las dependencias en modo producción:
     - npm install --production
  3. Configurar el servidor en modo producción:
     - export NODE_ENV=production (bash)
     - set NODE_ENV=production (Windows)
  4. Establecer el arranque de la aplicación:
     - node server.js
- Las utilidades se puede publicar en el repositorio npmjs.com para que se pueda instalar vía npm install.
- Es necesario disponer de una cuenta de usuario y seguir tres paso:
  1. Configurar los detalles en el archivo package.json
  2. Establecer las exclusiones en el archivo .npmignore
  3. npm publish

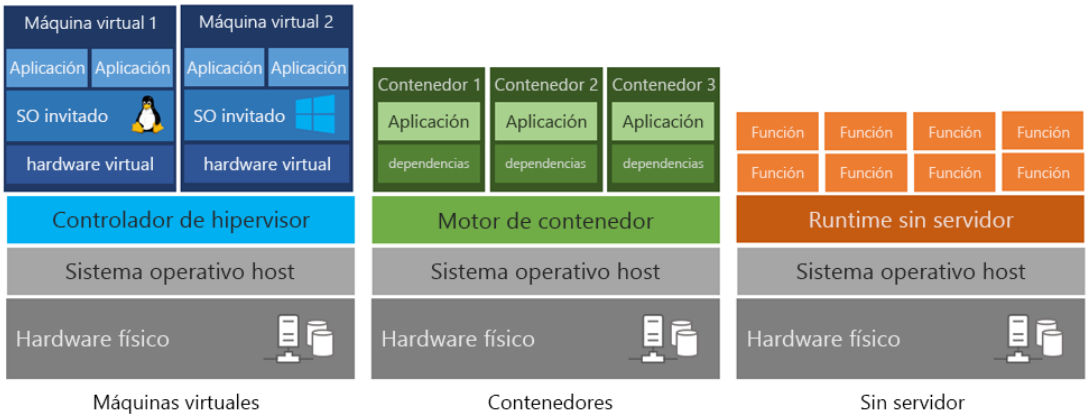
Modelo de despliegue

- El modelo de despliegue hace referencia al modo en que vamos a organizar y gestionar los despliegues de los microservicios, así como a las tecnologías que podemos usar para tal fin.
- El despliegue de los microservicios es una parte primordial de esta arquitectura. Muchas de las ventajas que aportan, como la escalabilidad, son posibles gracias al sistema de despliegue.
- Existen convencionalmente varios patrones en este sentido a la hora de encapsular microservicios:
  - Máquinas virtuales.
  - Contenedores.
  - Sin servidor: FaaS (Functions-as-a-Service)
- Los microservicios están íntimamente ligados al concepto de contenedores (una especie de máquinas virtuales ligeras que corren de forma independiente, pero utilizando directamente los recursos del host en lugar de un SO completo). Hablar de contenedores es hablar de Docker. Con este software se pueden crear las imágenes de los contenedores para después crear instancias a demanda.

Modelo de despliegue

- Las imágenes Docker son como plantillas. Constan de un conjunto de capas y cada una aporta un conjunto de software a lo anterior, hasta construir una imagen completa.
- Por ejemplo, podríamos tener una imagen con una capa Ubuntu y otra capa con un servidor LAMP. De esta forma tendríamos una imagen para ejecutar como servidor PHP.
- Las capas suelen ser bastante ligeras. La capa de Ubuntu, por ejemplo, contiene algunos los ficheros del SO y otros, como el Kernel, los toma del host.
- Los contenedores toman una imagen y la ejecutan, añadiendo una capa de lectura/escritura, ya que las imágenes son de sólo lectura.
- Dada su naturaleza volátil (el contenedor puede parar en cualquier momento y volver a arrancarse otra instancia), para el almacenamiento se usan volúmenes, que están fuera de los contenedores.

Contenedores



Modelo de despliegue

- Sin embargo, esto no es suficiente para dotar a nuestro sistema de una buena escalabilidad. El siguiente paso será pensar en la automatización y orquestación de los despliegues siguiendo el paradigma cloud. Se necesita una plataforma que gestione los contenedores, y para ello existen soluciones como Kubernetes.
- Kubernetes permite gestionar grandes cantidades de contenedores, agrupándolos en pods. También se encarga de gestionar servicios que estos necesitan, como conexiones de red y almacenamiento, entre otros. Además, proporciona también esta parte de despliegue automático, que puede utilizarse con sus componentes o con componentes de otras tecnologías como Spring Cloud+Netflix OSS.
- Todavía se puede dar una vuelta de tuerca más, incluyendo otra capa por encima de Docker y Kubernetes: Openshift. En este caso estamos hablando de un PaaS que, utilizando Docker y Kubernetes, realiza una gestión más completa y amigable de nuestro sistema de microservicios. Por ejemplo, nos evita interactuar con la interfaz CLI de Kubernetes y simplifica algunos procesos. Además, nos provee de más herramientas para una gestión más completa del ciclo de vida, como construcción, test y creación de imágenes. Incluye los despliegues automáticos como parte de sus servicios y, en sus últimas versiones, el escalado automático.
- Openshift también proporciona sus propios componentes, que de nuevo pueden mezclarse con los de otras tecnologías.

FaaS (Functions-as-a-Service)

- El auge de la informática sin servidor es una de las innovaciones más importantes de la actualidad. Las tecnologías sin servidor, como Azure Functions, AWS Lambda o Google Cloud Functions, permiten a los desarrolladores centrarse por completo en escribir código. Toda la infraestructura informática de la que dependen (máquinas virtuales (VM), compatibilidad con la escalabilidad y demás) se administra por ellos. Debido a esto, la creación de aplicaciones se vuelve más rápida y sencilla. Ejecutar dichas aplicaciones a menudo resulta más barato, porque solo se le cobra por los recursos informáticos que realmente usa el código.
- La arquitectura serverless habilita la ejecución de una aplicación mediante contenedores efímeros y sin estado; estos son creados en el momento en el que se produce un evento que dispare dicha aplicación. Contrariamente a lo que nos sugiere el término, serverless no significa «sin servidor», sino que éstos se usan como un elemento anónimo más de la infraestructura, apoyándose en las ventajas del cloud computing.
- La tecnología sin servidor apareció por primera vez en lo que se conoce como tecnologías de plataforma de aplicaciones como servicio (aPaaS), actualmente como FaaS (Functions-as-a-Service).

Patrones de despliegue

- Multiple service instances per host
  - Ejecutar varias instancias de diferentes servicios en un host (máquina física o virtual).
- Service instance per host
  - Implementar cada instancia de servicio individual en su propio host
- Service instance per VM
  - Empaquetar el servicio como una imagen de máquina virtual e implementar cada instancia del servicio como una VM separada
- Service instance per Container
  - Empaquetar el servicio como una imagen de contenedor (Docker) e implementar cada instancia del servicio como un contenedor
- Serverless deployment
  - Utilizar una infraestructura de implementación que oculte cualquier concepto de servidores (es decir, recursos reservados o preasignados), hosts físicos o virtuales, o contenedores. La infraestructura toma el código del servicio y lo ejecuta. Se factura por cada solicitud en función de los recursos consumidos.
- Service deployment platform (API Management)
  - Utilizar una plataforma de implementación, que es una infraestructura automatizada para la implementación de aplicaciones.

Despliegue

- Crear el fichero "dockerfile":

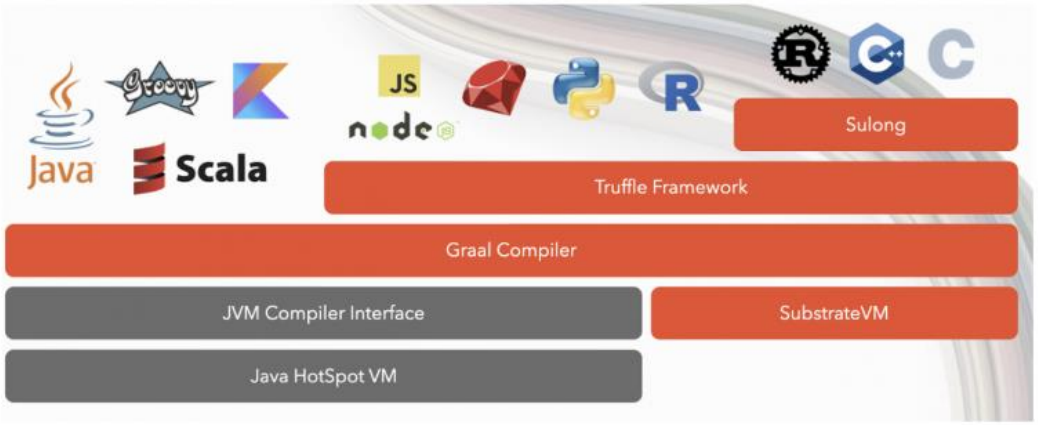
```
FROM node:lts-alpine
ENV NODE_ENV=production
WORKDIR /app
COPY ["package.json", "package-lock.json*", "npm-shrinkwrap.json*", "./"]
RUN npm install --production --silent && mv node_modules ../
COPY . .
EXPOSE 4321
VOLUME [ "/app/uploads", "/app/public", "/app/data", "app/log" ]
CMD [ "node", "server.js" ]
```

- Crear imagen:
  - docker build -t node-server .
- Crear y ejecutar contenedor:
  - docker run -d --name node-server -p 8080:4321 node-server

GraalVM

- GraalVM, es una máquina virtual creada por Oracle Labs como podría serlo la tradicional JVM. No obstante, cabe destacar algunas mejoras. En primer lugar, es políglota, es decir, una máquina virtual capaz de ejecutar código en diversos lenguajes de programación. Por otro lado, incorpora un nuevo concepto de compilador de tipo AOT (ahead of time) que predice todo lo que nuestro código necesitará para ejecutarse lo que permite la creación de imágenes nativas.
- GraalVM es un conjunto de herramientas que puede ser utilizado de diversas formas:
  - Graal compiler puede entenderse como un compilador de código Java tradicional, con la diferencia que podemos “elegir” usar JIT (just in time) o el nuevo AOT (ahead of time).
  - SubstrateVM es un runtime necesario para ejecutar el AOT compiler de la JVM y generar las imágenes nativas.
  - Truffle es un framework usado dentro de GraalVM como interprete de otros lenguajes como Ruby, R, Python, entre otros. C/C++, Fortran y otros requieren de Sulong.
  - GraalVM: el paquete completo de tecnologías que puede ser utilizado para diversos casos de uso, ejecutar código Java compilado con JIT, con AOT, ejecutar otros lenguajes, ejecutar código mezclando lenguajes, etc.
- GraalVM está disponible como ediciones GraalVM Enterprise, basada en Oracle JDK y con optimizaciones adicionales además del soporte, y GraalVM Community, basada en OpenJDK, e incluyen soporte para Java 11 y Java 17. Están disponibles para Linux y macOS en sistemas x86 de 64 bits y ARM de 64 bits, y para Windows en sistemas x86 de 64 bits.

Arquitectura GraalVM



GraalVM JavaScript

- Una implementación de alto rendimiento del lenguaje de programación JavaScript. Construido sobre GraalVM por Oracle Labs.
- Los objetivos de GraalVM JavaScript son:
  - Ejecutar código JavaScript con el mejor rendimiento posible
  - Compatibilidad total con la última especificación ECMAScript
  - Admitir aplicaciones Node.js, incluidos paquetes nativos
  - Permitir una actualización sencilla desde aplicaciones basadas en Nashorn o Rhino
  - Rápida interoperabilidad con Java, Scala o Kotlin, o con otros lenguajes GraalVM como Ruby, Python o R
  - Ser integrable en sistemas como Oracle RDBMS o MySQL

GraalVM JavaScript

- GraalVM JavaScript puede ejecutar aplicaciones Node.js. Proporciona alta compatibilidad con paquetes npm existentes, con una alta probabilidad de que su aplicación se ejecute de inmediato. Esto incluye paquetes npm con implementaciones nativas. Tenga en cuenta que algunos módulos npm deberán volver a compilarse desde el código fuente con GraalVM JavaScript si se envían con archivos binarios compilados para Node.js basado en V8.
- Al igual que el propio JavaScript, Node.js es un componente instalable por separado de GraalVM (desde 21.1). Se puede instalar usando GraalVM Updater:
  - $JAVA_HOME/bin/gu install nodejs
  - $JAVA_HOME/bin/node --version